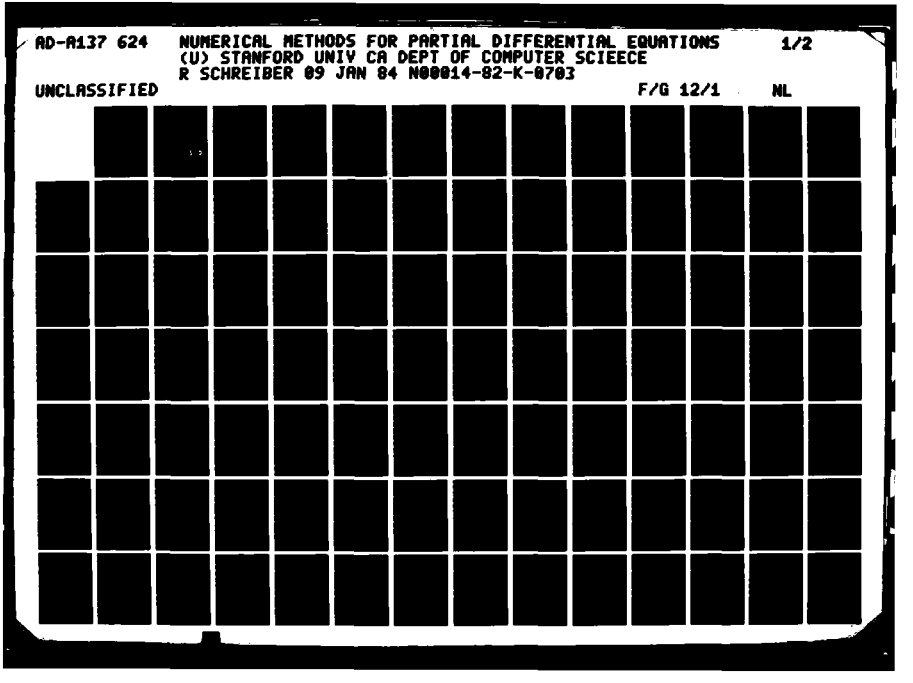
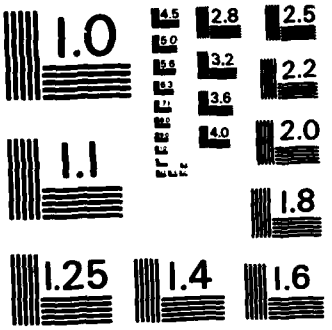

AD-A137 624 NUMERICAL METHODS FOR PARTIAL DIFFERENTIAL EQUATIONS 1/2
(U) STANFORD UNIV CA DEPT OF COMPUTER SCIEECE
R SCHREIBER 09 JAN 84 N00014-82-K-0703
UNCLASSIFIED F/G 12/1 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Final Report	2. GOVT ACCESSION NO. AD A137624	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Numerical Methods for Partial Differential Equations		5. TYPE OF REPORT & PERIOD COVERED Final Report, 8/15/82 - 8/14/83 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert Schreiber		8. CONTRACT OR GRANT NUMBER(s) N00014-82-K-0703
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department Stanford University Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Department of the Navy Arlington, VA 22217		12. REPORT DATE 1/9/84 13. NUMBER OF PAGES Three
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Scientific Officer, one copy
Administrative Contracting Officer, one copy
Director, Naval Research Laboratory, six copies
Defense Technical Information Center, twelve copies

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

DTIC
ELECTE
S FEB 8 1984 D
B

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473

84 01 17 044

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

FINAL REPORT
ONR CONTRACT N00014 - 82 - k - 0703
15 August 1982 - 14 August 1983
Robert Schreiber

The focus of research was the application of systolic array architectures to computations in numerical linear algebra, and the applications of these architectures to digital signal processing (DSP) and elliptic partial differential equations (PDE). Research was conducted on a number of topics; first I will discuss those for which complete reports have been written.

- Matrix triangularization by systolic arrays.
  Earlier work by Gentlemen and Kung was extended; many practical questions concerning application to problems in DSP were discussed [1]. The literature on factorization of banded matrices was unified and extended [6].

- Singular value and eigenvalue computations;
  A new architecture for singular value decomposition (SVD) was developed [3]. I also considered the use of a proposed systolic architecture for eigenvalue and SVD when the matrix is too large for the systolic array to accommodate [4,7].

- Elliptic PDE; and
  The design of a highly parallel architecture for the multigrid method and an analysis of its performance was given in joint work with T. Chan of Yale [2,8].

- Updating Cholesky factorizations.
  The problem is to recompute the Cholesky Factorization ($A = LL^T$) of a symetric positive definite matrix when it is changed by a matrix of low rank. This arises often in DSP and also in quasi-Newton methods in optimization. I described several systolic architectures in joint work with my student, W.P. Tang [5].

Several projects were begun, and work continues on these. During my stay at the Royal Institute of Technology in Stockholm I began work with Dainis Millars on construction of systolic arrays with custom VLSI and off-the-shelf VLSI components. A report is forthcoming; a patent for the custom chip will be sought.

Work also began in Stockholm, with Erik Tiden and Bjorn Lisper, on the synthesis and verification of systolic arrays.

With Lars Elden of Linkoping University, I developed a systolic architecture for linear, discrete ill-posed problems. The report will appear early in 1984.

Earlier work of mine on systolic methods for the eigenvalue problem raised the issue of pipelining iterations of the QR algorithm. The difficulty in doing so is in finding a suitable shift strategy. A student, W. Wilson, has begun some numerical experiments; his results, unfortunately, are not encouraging. A report will appear in 1984.

Finally, work has begun on the implementation of several modern high resolution direction-finding algorithms in DSP.

REFERENCES

1. R. Schreiber and P.J. Kuekes
Systolic linear algebra machines in digital signal processing.
In VLSI and Modern Signal Processing, S.Y. Kung, H.J. Whitehouse, and T. Kailath, editors. Prentice-Hall, 1984.

2. R. Schreiber
Systolic arrays: high performance parallel machines for Matrix computation.
In Elliptic Problem Solvers, A. Schoenstadt and G. Birkhoff, editors. Academic Press, 1984.

3. R. Schreiber
A systolic architecture for singular value decomposition.
EDF Bulletin de la Direction des Etudes et Recherches, Serie C, No.1, (1983) pp. 143-148.

4. R. Schreiber
On the systolic arrays of Brent, Luk. and van Loan for the symmetric eigenvalue and singular value problems.
Report TRITA - NA - 8311, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, 1983.
Submitted to SIAM Journal on Scientific and Statistical Computing.

5. R. Schreiber and W.P. Tang
On systolic arrays for updating the Cholesky Factorization.
Report TRITA-NA-8313, Department of Numerical Analysis and Computing Science. Royal Institute of Technology, Stockholm, Sweden, 1983.
Submitted to BIT.

6. R. Schreiber
On systolic array methods for band matrix factorizations.
Report TRITA-NA-8316, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, 1983.

7. R. Schreiber
On the systolic arrays of Brent, Luk, and Van Loan.
In Real-Time Signal Processing VI, Keith Bromley, editor, Proc. SPIE 431. pp. 72-76(1983).

8. R. Schreiber and Tony F. Chan
Parallel Networks for Multi-Grid Algorithms: Architecture and Complexity.
Report #262, Department of Computer Science, Yale University, 1983.
Submitted to SIAM Journal of Scientific and Statistical Computing.

A Systolic Architecture for Singular Value Decomposition

Une architecture systolique pour la décomposition en valeurs singulières

R. Schreiber
(Stanford, USA)

1 INTRODUCTION

Systolic arrays are highly parallel computing structures specific to particular computing tasks. They are well-suited for reliable and inexpensive implementation using many identical VLSI components. The designs consist of one and two-dimensional lattices of identical processing elements. Communication of data occurs only between neighboring cells. Control signals propagate through the array like data. These characteristics make it feasible to construct very large arrays.

Several modern methods in digital signal processing require real-time solution of some of the basic problems of linear algebra [13]. Fortunately systolic arrays have been developed for many of these problems [4,10,12]. But several gaps remain. Only partially satisfying results have been obtained for the eigenvalue and singular value decompositions, for example.

Here we consider a systolic array for the singular value decomposition (SVD). An SVD of an m x n ($m \geq n$) matrix A is a factorization

$$A = U \Sigma V^T$$

where U is m x n with orthonormal columns, $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$, and V is orthogonal. There are many important applications of the SVD [1,6,13].

There have been several earlier investigations of parallel SVD algorithms and arrays. First, Finn, Luk, and Pottle describe a systolic structure of $n^2/2$ processors and two algorithms that use it. But the convergence of their algorithms has not been proved and may be slow [3]. Heller and Ipsen [8] describe an array for computing the singular values of a banded matrix with bandwidth w. They use $O(w)$ processors and $O(wn^2)$ time. Brent and Luk [2] describe an n/2 processor linear array that implements a one-sided orthogonalization method and converges reliably in $O(n \log n)$ time. Unfortunately the processors in this array are quite complex, and it is not clear that matrices with more than n columns can be efficiently accomodated.

In this paper we discuss two topics. First, we show how an architecture for computing the eigenvalues of a symmetric matrix can be modified to compute singular values and vectors. Second, we discuss the implementation using VLSI chips of these systolic eigenvalue and SVD arrays.

The SVD is often used to regularize ill-conditioned problems. In these there are $p < n$ large singular values and n-p that are much smaller. What is needed is the pseudoinverse of the rank p matrix closest (with respect to the 2-norm) to A,

$$A_{(p)} = u_1 \sigma_1 v_1^T + \ldots + u_p \sigma_p v_p^T$$

We have recently developed a new algorithm to compute $A^+_{(p)}$ that involves nothing but a sequence of matrix-matrix products, for which systolic arrays are well-known (see, e.g., [9].) An alternate form of the algorithm can be used to compute the related orthogonal projection matrix

$$P_{(p)} = v_1 v_1^T + \ldots + v_p v_p^T$$

2 AN SVD ARCHITECTURE

Let A be a given matrix. The singular values of A will be obtained in two phases:

1. A is reduced to an upper triangular matrix B with bandwidth k+1,

   $b_{ij} = 0$ if $i < j$ or $i > j+k$,

   and $B = QAP$ where Q and P are orthogonal.

2. B is diagonalized by an iterative process equivalent to implicitly shifted QR iteration on B^TB.

With k=1 this is the standard method of Golub and Reinsch [7]. The reason for allowing k>1 is an increase in the parallelism. In phase 1, kn processors are employed; the time is O(mn/k). In phase 2, $2k^2$ processors are used; the time per iteration is 6n+O(k).

2.1 Reduction to banded form

The reduction step uses a k x n trapezoidal array that has been described in detail previously [12]. Let the m x n matrix X be partitioned as

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}$$

where X_{11} is k x k. The array applies a sequence of Givens rotations to the rows of X to zero the first k columns below the main diagonal. If Q is the product of these rotations, then

$$QX = \begin{bmatrix} R_{11} & Y_{12} \\ 0 & Y_{22} \end{bmatrix}$$

where R_{11} is k x k upper triangular. R_{11}, Y_{12}, Y_{22}, and the parameters of the rotations that make up Q all flow out from the array. The time required is m. (Here and below we give "times" in units of the time required for an individual cell in the array to carry out its computation.)

Now let

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

be the given matrix. Send A through the array to produce

$$Q_1 A = \begin{bmatrix} R_{11} & C_{12} \\ 0 & C_{22} \end{bmatrix}$$

Next send $[C_{12}^T, C_{22}^T]$ through to produce

$$P_1[C_{12}^T, C_{22}^T] = [L_{12}^T, \bar{A}_{22}^T]$$

(Although the input matrix has m columns, the array can handle this factorization in time m by making $\lceil m/n \rceil$ passes over the data [12]. Now continue this process using $\bar{A}_{22}$ in place of A. After $J = \lceil n/k \rceil$ such steps we have produced a k+1 diagonal, upper triangular matrix B,

$$B = \begin{bmatrix} R_{1,1} & L_{1,2} & & & \\ & R_{2,2} & L_{2,3} & & \\ & & \ddots & \ddots & \\ & & & & L_{J-1,J} \\ & & & & R_{J,J} \end{bmatrix}$$

such that A = QBP where Q and P are orthogonal. The total time used is mJ = mn/k.

The transposition of data required can be done by a specialized switching device, a "systolic shifter," described earlier [12].

When singular vectors are to be computed, the rotations generated by the array may be applied to identity matrices of order m and n. This can be done by the array. These matrices accumulate the product of the rotations used, that is the orthogonal matrices Q and P above.

2.2 QR iteration

Now we consider QR iteration to get the singular values of B, hence those of A. We shall generate a sequence of matrices $\{B^{(i)}\}$ having the same structure as B and converging to a diagonal matrix. $B^{(0)} = B$ and $B^{(i+1)} = P^{(i)}B^{(i)}Q^{(i)}$ where $P^{(i)}$ and $Q^{(i)}$ are orthogonal.

First we consider QR iteration on B^TB without shifts. This can be realized by the procedure

1. Find $Q^{(i)}$ such that
   $L^{(i)} = B^{(i)}Q^{(i)}$
   is lower triangular,
2. Find $P^{(i)}$ such that
   $B^{(i+1)} = P^{(i)}L^{(i)}$
   is upper triangular.

Both steps of this procedure can be carried out by the Heller-Ipsen (HI) array [8]. This is a k x w rectangular array for QR factorization of w-diagonal matrices. In this array, plane rotations are generated at the left edge and move to the right, affecting a pair of matrix rows. Take w = k+1. $B^{(i)}$ enters the matrix at the bottom, each diagonal entering, one element at a time, into one of the processors. The array annihilates the elements of the upper triangle of $B^{(i)}$. This causes fill-in of k diagonals in the lower triangle. The resulting matrix $L^{(i)}$ emerges from the top in the same diagonal-per-processor format. It immediately enters a second array. This array annihilates the lower triangle of $L^{(i)}$ and the resulting upper triangular matrix $B^{(i+1)}$ emerges from the top (Fig. 1) The time is 2n+4k per iteration: element a_{nn} enters the bottom array at time 2n, leaves at the upper left corner at time 2n+2k, and leaves the top array at time 2n+4k.

Unshifted QR converges slowly. The rate of convergence of b_{11} to σ_1 is σ_2^2/σ_1^2. In some situations this may be adequate and the simplicity of the structure used is then a real advantage. It is also easy to pipeline the iterations. As $B^{(i+1)}$ comes out of the second array it can be sent directly into another pair of arrays to begin the (i+1)th iterations, etc. As many as n/4k iterations can be effectively pipelined; any more and the pipe length exceeds n, so that the pipe never gets full. If we choose $k = O(n^{1/2})$ and pipeline $n/4k = O(n^{1/2})$ iterations then the number of processors in both arrays is $O(n^{3/2})$ and the total time, assuming O(n) iterations of QR are required, is also $O(n^{3/2})$. These considerations also apply to the array implementation of the implicitly shifted QR algorithm that is discussed below, with one important proviso. When pipelining the iterations, some strategy for choosing several shifts in advance must be used.

2.2.1 Implicitly shifted QR iteration

To obtain adequate convergence speed we need to incorporate shifts. Following Stewart [14], suppose that one QR iteration with shift τ is performed on B^TB, and the orthogonal matrix so generated is Q. Then proceed as follows:

1. Let Q_0 be any matrix whose first k columns are the same as those of Q;
2. Using the same technique as in Section 2.1, reduce BQ_0 to upper triangular k+1 diagonal form, yielding a matrix B'.

It can be shown that $B'^{T}B'$ is the matrix that would result from one QR step with shift λ applied to $B^{T}B$.

To use the trapezoidal array as described above to carry out step 2 would be inefficient. Rather we proceed as follows. Q_0^T is composed of plane rotations that zero the first k columns of $(B^TB-\lambda)$ below the main diagonal. Applying Q_0 to B causes fill-in in the k diagonals below the main diagonal, confined to rows 2, 3, ..., 2k. See Fig. 2 for the case k=2.

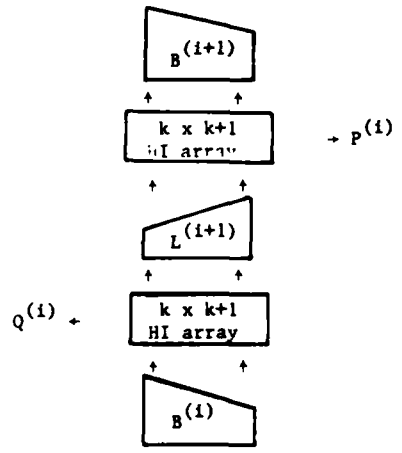


Figure 1

Unshifted QR iteration with two Heller-Ipsen arrays

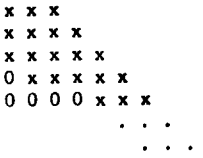


Figure 2

Structure of BQ_0, k=2

Let the first 2k rows of BQ_0 be sent into a k x 2k+1 HI array. By a sequence of plane rotations applied to the rows, the array removes the "bulge" in the lower triangle, adding a bulge of the same shape in the first 3k columns of the upper triangle. This data flows directly into another k x 2k+1 HI array that removes the elements to the right of the k^{th} superdiagonal and causes a new bulge to appear in the lower triangle, in columns k+1 through 3k-1 and extending to row 3k. (The second HI array is the mirror image of the first. Rotations are generated at its right edge and move left, affecting pairs of matrix columns. Let P_1 and Q_1 be the orthogonal matrices implicitly used by the two HI arrays. The matrix emerging from the second array is

$$B_1 = P_1 B Q_0 Q_1$$

and it has the form

$$B_1 = \begin{bmatrix} B_{11} \\ 0 \quad B_{21} \end{bmatrix}$$

where B_{11} is a k x n upper triangular, k+1 diagonal matrix, and B_{21} is an n-k x n-k matrix of the same form as BQ_0. Fig. 3 illustrates this for k=2.

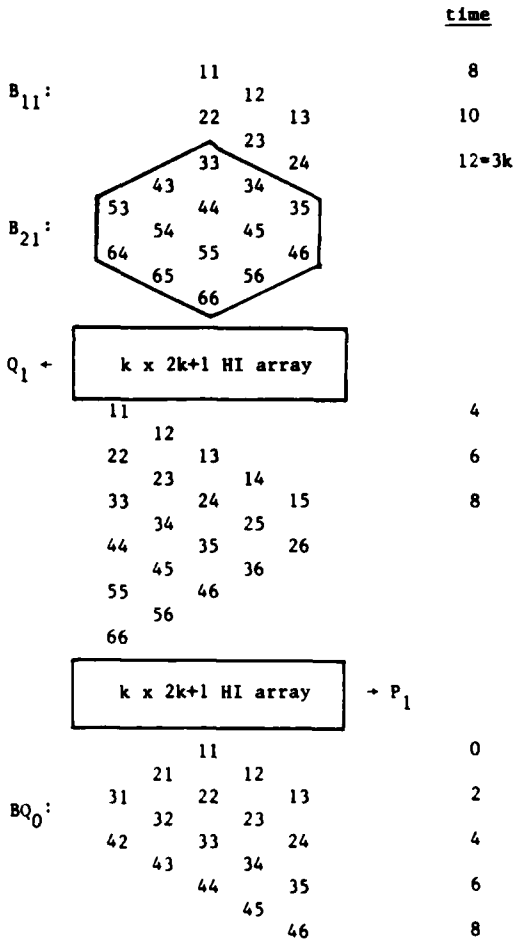


Figure 3

"Chasing the bulge" with two k x 2k+1 Heller-Ipsen arrays

Now we do exactly the same thing to B_{21}, etc. This yields matrices

$$B_j = P_j B_{j,j-1} Q_j, \qquad j=2,\ldots,J$$

with

$$B_j = \begin{bmatrix} B_{j,j} & \\ 0 & B_{j+1,j} \end{bmatrix}$$

and $J = \lceil (n-1)/k \rceil$. Finally $B' = P_J \ldots P_1 B Q_0 \ldots Q_J$ is the matrix we require.

The time neede is 6n. It takes 2k steps for an HI array to start producing output. Thus, the second array starts its output at time 4k. The first element of $B_{j+1,j}$, which is the $(k+1)^{st}$ element of the main diagonal to come out of the second array, comes out at time 6k. By this time the first arrays inputs have become idle, so this element can immediately reenter. Therefore one step, from B_j to B_{j+1}, takes time 6k. There are $\lceil (n-1)/k \rceil$ such steps, hence about 6n time for the whole process.

2.3 Complex matrices

In signal processing applications, complex matrices often arise. Here we discuss the algorithms to be used for QR iteration with complex matrices. Essentially we show that the plane rotations used can be of a special form:

$$(1) \quad \begin{aligned} x' &= c^*x + \sigma y \\ y' &= -\sigma x + cy \end{aligned}$$

where x, y, and c are complex and σ is real. This saves 1/4 of the multiplications used by a fully complex plane rotation with complex s instead of σ -- 12 are used instead of 16. We shall call these c,σ rotations.

It is possible to compute the SVD of a complex m x n matrix $A = A_R + iA_I$ using real arithmetic. One finds the SVD of the 2m x 2n real matrix

$$\begin{bmatrix} A_R & -A_I \\ A_I & A_R \end{bmatrix}$$

Among the 2n singular values each singular value of A occurs twice, and the singular vectors are of the form $[x_R^T, x_I^T]$ where $x = x_R + ix_I$ is a singular vector of A. But the cost is much greater. In units where the cost of doing an m x n real SVD is one, the cost for the real 2m x 2n SVD is 8 while the complex m x n approach costs 3 (not 4, since the use of the c,σ rotations saves 1/4 of the work).

We now show that the c,σ rotations suffice. To start, we note that the banded matrix B produced by the reduction phase can always be chosen to have positive real elements on its main and k^{th} superdiagonals. Indeed the reduction $B = QAP$ to k+1 diagonal, upper triangular form is not unique:

$$B = QD_1 \, (D_1^{-1} A D_2^{-1}) \, D_2 P$$

is also such a reduction for any unitary diagonal matrices D_1 and D_2. These can always be chosen to give B the stated property. In fact, the trapezoidal array can do this automatically [12]. When it generates a rotation to zero some matrix element, the second element of the pair (x,y) for instance, it chooses the paramenters so that the result of the rotation is the pair

$$((|x|^2 + |y|^2)^{1/2} \, , \, 0)$$

Furthermore, the elements to be zeroed are the real elements resulting from previous rotations. The rotations to do the zeroing can, for this reason, be taken to be c,σ rotations.

Now we look at the second phase. Because of the structure of B, the main, k^{th} super and k^{th} subdiagonals of B^TB are all real. The rotations that comprise Q_0 can be taken to be c,σ rotations since they zero real elements. And by keeping track of the locations of real elements one can show that in BQ_0 all elements of the outer diagonals are real. Again because the elements to be annihilated are real, c,σ rotations can be used to eliminate the bulge. A matrix with the same structure as BQ_0 results, and the proof therefore follows by induction.

2.4 An alternate scheme

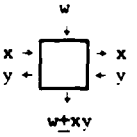
Gene Golub has pointed out that the eigenvalues of the 2n x 2n matrix

$$C = \begin{bmatrix} 0 & B \\ B^T & 0 \end{bmatrix}$$

are the singular values of A taken with positive and negative sign, and if (x^T, y^T) is an eigenvector of C then x is a left singular vector of B and y is a right singular vector of B [5]. Thus we may attempt to find the eigendecomposition of C. After a symmetric interchange of rows and columns corresponding to the permutation (n+1, 2, n+2, 2, ..., 2n, n), C is a symmetric 4k-1 diagonal matrix. A 2k-1 x 4k-1 HI array can implement one step of the QR method with shifts for this matrix in $n + O(k)$ time [10]. In the complex case, both C and the permuted C have real outermost diagonals, so c,σ rotations can be used. Thus, although twice as much hardware is used, the time per iteration is 1/6 as great as for the previous scheme.

3 VLSI IMPLEMENTATION

Now we consider how to build the cells of the HI array. The fundamental unit we use in this construction is a multiply-add cell, whose function is this:

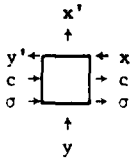


Outputs leave the cell one clock after inputs enter.

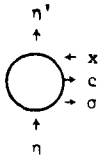
Although other primitive units (CORDIC blocks, for example) might be used, we feel that the multiply-add is a good basis for such an investigation. Currently, a floating point multiply-add is about what can be integrated on a single chip. It is almost universally useful. Indeed, the multiply-add pair is often the inner loop in numerical linear algebraic computations. Even when larger cells and pieces of arrays can be integrated into single chips, designs based on the multiply-add primitive will be useful.

We shall discuss implementation of the HI array cells for complex data. The real case was discussed earlier [11] as were the cells of the trapezoidal array [12].

The complex HI array triangularizes a banded input matrix using c,σ rotations of the form (1). The rotations are applied to a pair (x,y) of matrix elements by an internal cell



after having been generated by a boundary cell



by

$$x' = (\eta^2 + |x|^2)^{1/2}$$
$$c = x / x'$$
$$\sigma = \eta / x'$$

In the internal cell computation, 4 quantities are computed, each requiring 3 multiplies and 2 adds. Let z_R and z_I denote the real and imaginary parts of the complex quantity z. The computed values are

$$x'_R = c_R x_R + c_I x_I - \sigma y_R,$$
$$x'_I = c_R x_I - c_I x_R - \sigma y_I;$$
$$y'_R = c_R y_R - c_I y_I + \sigma x_R,$$
$$y'_I = c_R y_I + c_I y_R + \sigma x_I.$$

Using 4 multiply-add chips we can construct a compound cell that gives these results in the least possible time, 3 clocks. We assume that complex quantities are represented in "word serial" form, with the real part preceding the imaginary part on the same data path. A schedule using 4 chips that achieves the minimum latency is shown in Table 1.

Table 1. Schedule for Internal HI Cell

time	Chip C1	Chip C2	Input/Output c_i	Input/Output y	Input/Output x	Chip C3	Chip C4
0			c_R				
1	$c_R y_R$		c_I	y_R	x_R	$c_R x_R$	
2	$c_I y_R$	$-c_I y_I$	σ	y_I	x_I	$-c_I x_R$	$c_I x_I$
3	σx_R	$c_R y_I$	c_R			$-\sigma y_R$	$c_R x_I$
4		σx_I	c_I	y'_R	x'_R		$-\sigma y_I$
5			σ	y'_I	x'_I		

The computation at the boundary cell is this: given inputs x and η^2, compute

$$\eta'^2 = \eta^2 + x_R^2 + x_I^2$$
$$\eta' = \sqrt{\eta'^2}$$
$$c_R = x_R / \eta'$$
$$c_I = x_I / \eta'$$
$$\sigma = \eta / \eta'$$

A second primitive, for divide and square root, is needed to implement the boundary cell. We assume that a chip for computing

$$(a,b) \Longrightarrow a / b^{1/2}$$

is available. A compound cell using one multiply-add and two of these square root chips can produce results at the rate required to keep up with the internal cell. A schedule is shown in Table 2. The overall array timing is now that of the "ideal" HI array in which everything happens in a single cycle (of length 3 chip clocks) . The cells are used 1/2 of the time, but two independent problems can be solved simultaneously, making full use of the hardware.

Table 2. Schedule for HI Boundary Cell

time	Chips: mult-add	Chips: sqrt #1	Chips: sqrt #2	I/O
1	$\rho^2 + x_R^2$			ρ^2, x_R
2	$+x_I^2$ $(=\rho'^2)$			x_I
3		$x_R[\rho'^2]^{-1/2}$		
4		$x_I[\rho'^2]^{-1/2}$		ρ'^2, c_R
5		$\sigma[\rho'^2]^{-1/2}$	$[\rho'^2]^{-1/2}$	c_I
6				σ, ρ'

ACKNOWLEDGEMENT

This research was partially supported by the Office of Naval Research under Contract N00014-82-K-0703 and by ESL, Incorporated, Sunnyvale, Calif.

REFERENCES

/1/ H. C. Andrews and C. L. Patterson : "Singular Value Decomposition and Digital Image Processing", IEEE Trans. Acoustics, Speech, and Signal Processing ASSP-24 (1976), pp. 26-53.

/2/ Richard P. Brent and Franklin T. Luk : "A Systolic Architecture for the Singular Value Decomposition", Report TR-CS-82-09, Department of Computer Science, The Australian National University, Canberra, 1982.

/3/ Alan M. Finn, Franklin T. Luk, and Christopher Pottle : "Systolic Array Computation of the Singular Value Decomposition", Real Time Signal Processing V, SPIE Vol. 341, Bellingham Wash., Society of Photo-optical Instrumentation Engineers, 1982.

/4/ W. M. Gentleman and H. T. Kung : "Matrix Triangularization by Systolic Array", Real Time Signal Processing IV, SPIE Vol. 298, Bellingham, Wash., Society of Photo-optical Instrumentation Engineers, 1981.

/5/ Gene Golub. Private communication.

/6/ G. H. Golub and F. T. Luk : "Singular Value Decomposition: Applications and Computations", ARO Report 77-1, Trans. of the 22nd Conf. of Army Mathematicians (1977), pp. 577-605.

/7/ G. H. Golub and C. Reinsch : "Singular Value Decomposition and Least Squares Solutions", Numer. Math. 14, (1970), pp. 403-420.

/8/ Don E. Heller and Ilse C. F. Ipsen : "Systolic Networks for Orthogonal Decompositions, with Applications", SIAM J. Scient. and Stat. Comput., to appear.

/9/ H. T. Kung and Charles Leiserson : "Systolic Arrays for (VLSI)", in Carver Mead and Lynn Conway, Introduction to VLSI Systems, Reading, Mass., Addison-Wesley, 1980.

/10/ Robert Schreiber : "Systolic Arrays for Eigenvalue Computation", Real Time Signal Processing V, SPIE Vol. 341, Bellingham, Wash., Society of Photo-optical Instrumentation Engineers, 1982

/11/ Robert Schreiber : "Systolic Arrays for Eigenvalues", Proc. of the Inter-American Workshop in Numerical Analysis, New York, Springer-Verlag, to appear.

/12/ Robert Schreiber and Philip J. Kuekes : "Systolic Linear Algebra Machines in Digital Signal Processing", in Sun-Yuan Kung, ed., Proc. of the USC Workshop on VLSI and Modern Signal Processing, Englewood Cliffs, New Jersey, Prentice-Hall, to appear.

/13/ J. M. Speiser and H. J. Whitehouse : "Architectures for Real-Time Matrix Operations", Proc. Government Microcircuit Applications Conf., held in Houston, Tex., 1980.

/14/ G. W. Stewart : Introduction to Matrix Computations, New York, Academic Press, 1973.

Abstract

We describe and analyse a family of highly parallel special purpose computing networks that implement multi-grid algorithms for solving elliptic difference equations. The networks have many of the features and advantages of systolic arrays. We consider the speedup achieved by these designs and how this is affected by the choice of algorithm parameters and the level of parallelism employed. We find, for example, that when there is one processor per grid-point, the designs cannot avoid suffering a loss of efficiency as the grid-size tends to zero.

1. Introduction

We shall describe and analyse a family of highly parallel special-purpose computing networks that implement multi-grid algorithms for solving elliptic difference equations. These networks have the same characteristics - regularity, local communciation, and repetitive use of a single, simple processing element - that make systolic architectures attractive [9]. These architectural advantages make it possible to build large computing networks of VLSI cells that would be relatively cheap, reliable, and very powerful.

Parallel Networks for Multi-Grid Algorithms: Architecture and Complexity

Tony F. Chan[1]
and Robert Schreiber[2]

Report # 262

September 19, 1983

[1]Box 2158, Yale Station, Computer Science Dept., Yale Univ., New Haven, CT 06520. This work was supported in part by Department of Energy Grant DE-AC02-81ER10996 and completed while this author was a visitor at Institute National Recherche en Informatique et Automatique, Le Chesnay, France in 1983.

[2]Computer Science Dept., Stanford Univ., Ca 94305. This work was supported in part by the Office of Naval Research under contract number N00014-82-K-0703.

Both a basic and a full multi-grid algorithm are considered. The basic method reduces the error in a given initial approximation by a constant factor in one iteration. The full method requires no initial guess and produces a solution with error proportional to the truncation error of the discretization. These algorithms are representative of many variants of linear and nonlinear multi-grid algorithms.

The analysis assumes that we are solving a linear system originating in a discretization of an elliptic partial differential equation on a rectangle in R^d, using a regular n^d point grid. The network is a system of grids of processing elements. For each $1 \leq k \leq K$, processor grid P_k has $(n_k)^\gamma$ elements, where γ is an integer less than or equal to d, and $n_K = n$. The machine implements a class of multi-grid algorithms using a corresponding system of nested point grids. For each $1 \leq k \leq K$, point grid G_k has $(n_k)^d$ points. The key assumption, which is quite realistic, is that it takes this machine $O((n_k)^{d-\gamma})$ time to carry out the computation required by one step of the multi-grid algorithm on point grid G_k using processor grid P_k.

We shall consider the efficiency of these parallel implementations, defining efficiency to be the ratio of the speedup achieved to the number of processors employed [8]. We shall consider a design to be efficient if this ratio remains bounded by a positive constant from below as $n \to \infty$. The analysis will show that when $\gamma < d$ some algorithms can be efficiently implemented. But when $\gamma = d$ (this is the most parallelism one can reasonably attempt to use) no algorithm can be efficiently implemented. There does exist, in this case, one group of algorithms for which the efficiency falls off only as $(\log n)^{-1}$.

The analysis assumes that we implement the same algorithms used by uniprocessor systems. Convergence results for these algorithms have been rather well-developed recently [1, 5, 7]. We make no attempt to develop algorithms that exhibit concurrent operation on several grids. Note, however, that some encouraging experimental results with such an algorithm have been obtained recently by Gannon and Van Rosendale [6].

In any discussion of the practical use of a specialized computing device, it must be acknowledged that overspecialization can easily make a design useless. At least, the designed device should be able to solve a range of size of problems of a particular structure, perhaps solving large problems by making several passes over the data, solving a sequence of smaller subproblems, or with some other techniques. We shall consider how a large grid, with $(mn)^d$ points, can be handled by a system of processor grids with n^γ elements each having $O(m^d n^{d-\gamma})$

memory cells. Problems on nonrectangular domains can be handled by techniques requiring repeated solutions on either rectangular subdomains or containing domains.

Brandt [3] has also considered parallel implementations of multi-grid methods. He discusses the use of various interconnection networks and appropriate smoothing iterations. One of our results, a $(\log n)^2$ time bound for fully parallel, full multi-grid algorithms, is also stated in his paper.

2. Multi-Grid Algorithms

We shall consider multi-grid algorithms in a general setting. The continuous problem is defined by the triple $\{H, a(u,v), f(v)\}$, where H is a Hilbert space with a norm $\|.\|$, $a(u,v)$ is a continuous symmetric bilinear form on H x H, and $f(v) : H \rightarrow R$ is a continous linear functional. The problem is:

$$\text{Find } u \in H \text{ such that } \quad a(u,v) = f(v) \text{ for all } v \in H. \tag{1}$$

It can be shown that if $a(*,*)$ satisfies certain regularity conditions (for example, that $a(v,v) \geq c_0 \|v\|^2$ for all $v \in H$), then Problem (1) has a unique solution [4].

We consider finite dimensional approximations of Problem (1). Let M_j, $j \geq 1$, be a sequence of N_j-dimensional spaces, on which one can define a corresponding bilinear form $a_j(u,v)$ and a corresponding continuous linear functional $f_j(v)$, which are constructed to be approximations to $a(u,v)$ and $f(v)$ respectively. Also, since the multi-grid algorithms involve transferring functions between these spaces, we have to construct extension (interpolatory) operators $E_j : M_{j-1} \rightarrow M_j$.

We shall give two multi-grid algorithms, namely BASICMG and FULLMG, with FULLMG calling BASICMG in its inner loop. The two algorithms differ in that BASICMG starts its computation on the finest grid and works its way down to the coarser grids, whereas FULLMG starts with the coarsest grid and works its way up to the finest grid. In the conventional single processor case, BASICMG reduces the error on a certain grid by a *constant factor* in optimal time, whereas FULLMG reduces the error to *truncation error level* in optimal time.

We give the basic multi-grid algorithm BASICMG in Table (2-1). This is a recursive algorithm, although in practice it is usually implemented in an iterative fashion. The iterations are controlled by the predetermined parameters (c,j,m). In this sense it is a direct method, unlike related adaptive algorithms which control the iterations by examining relative changes in the residuals [2]. Figure (2-1) illustrates the iteration sequence in the case $c = 2$. The major

computational work is in the smoothing sweeps (subroutine SMOOTH), which usually consists of some implementation of the successive-over-relaxation or Jacobi iteration or the conjugate gradient method. The smoothing sweeps are used to annihilate the highly oscillatory (compared to the grid spacing) components of the error in z efficiently. We require that a suitably "parallel" method, Jacobi or odd-even SOR for example, be used as the smoother. In the next section, we shall discuss new architectures for implementing these smoothing operations in more efficient ways.

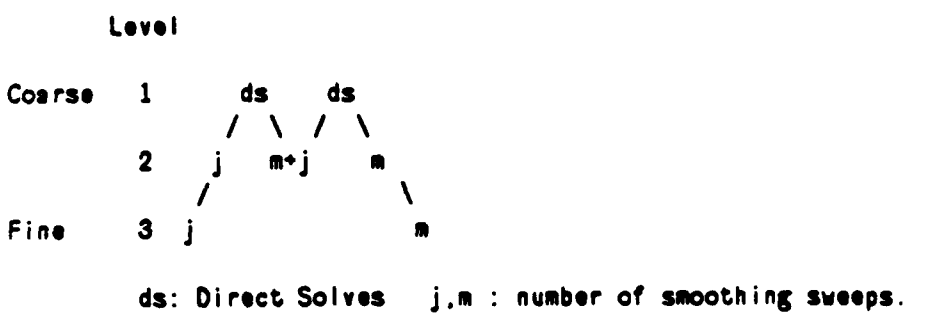
Table 2-1: BASICMG Algorithm

Algorithm BASICMG(k,z,c,j,m,a_k,f_k)

```
<Computes an approximation to u_k ∈ M_k,
where a_k(u_k,v) = f_k(v) for all v ∈ M_k,
given an initial guess z ∈ M_k.
Returns the approximate solution in z.
Reduces initial error in z by a constant factor.>

  If k = 1 then
    Solve the problem using a direct method. Return solution z.
  else
    <Smoothing step (j sweeps):>
      z ⇐ SMOOTH(j,z,a_k,f_k).

    <Form coarse grid correction equation:>
      a_{k-1}(q,v) = b_{k-1}  for all v ∈ M_{k-1}.

    <Solve coarse grid problem approximately by c cycles of BASICMG:>
      q ⇐ 0.
      Repeat c times:
        BASICMG(k-1,q,c,j,m,a_{k-1},b_{k-1})

    <Correction step:>
      z ⇐ z + E_k q.

    <Smoothing step (m sweeps):>
      z ⇐ SMOOTH(m,z,a_k,f_k).
  End If
End BASICMG
```

We give the full multi-grid algorithm FULLMG in Table (2-2). In the BASICMG algorithm, the choice of initial guess for u_k is not specified. In practice, good initial guesses are sometimes available essentially free (for example, from solutions of a nearby problem, from solutions at a previous time step, etc.). The FULLMG algorithm interpolates approximate solutions on coarser

Figure 2-1: Iteration of BASICMG for k = 3 and c = 2



grids as intial guesses for the BASICMG algorithm. It is also recursive and non-adaptive. Figure (2-2) illustrates the iteration sequence in the case k = 3, c = 2 and r = 1.

Table 2-2: FULLMG Algorithm

Algorithm FULLMG(k,z_k,r,c,j,m,a_k,f_k)

<Computes an approximation z_k to $u_k \in M_k$
where $a_k(u_k,v) = f_k$ for all $v \in M_k$,
using r iterations of BASICMG,
using initial guess from interpolating the approximate
solution obtained on the next coaser grid.
Solution obtained can be proven to have truncation error accuracy.>

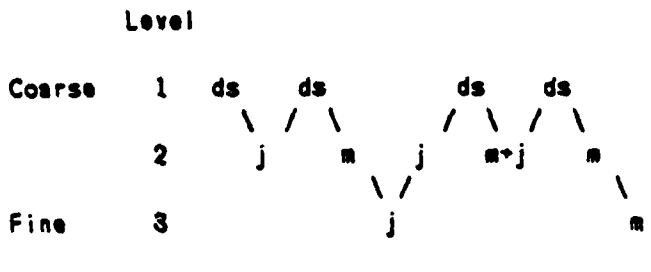
If k = 1 then
  Solve the problem using a direct method to get z_1.
else
  <Obtain solution on next coarser grid:>
    FULLMG($k-1,z_{k-1},r,c,j,m,a_{k-1},f_{k-1}$).

  <Interpolate z_{k-1}:>
    $z_k \Leftarrow E_k\, z_{k-1}$.

  <Reduce the error by iterating BASICMG r times:>
    Repeat r times:
      BASICMG(k,z_k,c,j,m,a_k,f_k).
End if
End FULLMG

We would like to summarize briefly the accuracy and convergence behaviour of the above two multi-grid algorithms. Since the main emphasis of this paper is on the algorithmic aspects of these multi-grid algorithms, we shall refer the reader to the literature for more details. The framework presented here is based on the work of and Bank and Dupont [1] and Douglas [5].

Figure 2-2: Iteration of FULLMG for k = 3, c = 2 and r = 1



The accuracy and the convergence of the BASICMG algorithm obviously depend on the three crucial steps of the algorithm: smoothing, coarse grid transfer, and fine grid correction. The basic requirements are that the smoothing sweeps annihilate the high frequency components of the error efficiently, the coarse grid correction q be a good approximation to the fine grid error in the low frequency components, and the interpolation operators (E_j's) be accurate enough. These conditions can be formalized into mathematically precise hypotheses which can then be verified for specific applications [5]. Assuming these hypotheses, one can show that Algorithm BASICMG reduces the error on level k by a constant factor provided that enough smoothing sweeps are performed. Moreover, it can be shown (see Section 4) that Algorithm BASICMG (for small values of c) can achieve this in optimal time, i.e. $O(N_k)$ arithmetic operations. Obviously, the work needed depends on the accuracy of the initial guess and increases with the level of accuracy desired. Often, one is satisfied with *truncation error accuracy*, i.e. $\|z - u\| = O(\|u_k - u\|) \leq C N_k^{-\ell}$ for some fixed ℓ and C which are independent of k. For a general initial guess, the straightforward application of Algorithm BASICMG to reduce the initial error to this level takes $O(N_k \log(N_k))$ time, which is not optimal. The FULLMG algorithm overcomes this problem by using accurate intial guesses obtained by interpolating solutions from coarser grids. The convergence result for Algorithm BASICMG can be combined with the basic approximation properties of the various finite dimensional approximations $\{M_j, a_j, f_j\}$ to show that Algorithm FULLMG computes a solution z_k that has truncation error accuracy in $O(N_k)$ time.

3. The Computing Network

In this section we describe a simple parallel machine design for multi-grid iteration. We restrict attention to linear elliptic problems in d dimensions over rectangular domains, to discretizations based on grids of n^d points, and to multi-grid methods based on a system of point-grids $\{G_k\}_{k=1}^K$ where G_k has $(n_k)^d$ gridpoints, with mesh lengths $h_{k,j}$, $1 \leq j \leq d$, the finest grid has $n_K = n$, and

$$n_{k+1} = a(n_k + 1) - 1, \quad k = 1, 2, \ldots, K-1$$

for some integer $a \geq 2$.

The machine consists of a system of processor-grids $\{P_k\}_{k=1}^K$ corresponding to the point-grids. Each processor-grid is an $(n_k)^\gamma$ lattice in which a processor is connected to its 2γ nearest neighbors.

We shall employ a standard multi-index notation for gridpoints and processors. Let

$$s_n^+ = \{0, 1, \ldots, n-1\}.$$

Let $s_{n,s}^+ = (s_n^+)^s$, the set of s-tuples of nonnegative integers less than n. We shall make use of a projection operator $\pi_r^s : s_{n,r}^+ \to s_{n,s}^+$ defined for $r \geq s$ by

$$\pi_r^s((i_1, \ldots, i_r)) = (i_1, \ldots, i_s).$$

By convention, if $\underline{i} \in s_{n,s}^+$, then $\underline{i} = (i_1, \ldots, i_s)$. Also let $\underline{1} = (1, \ldots, 1)$. We shall also use the norm $|\underline{i}| = |i_1| + \ldots + |i_s|$ on $s_{n,s}^+$.

We shall label the gridpoints in G_k with elements of $s_{n_k,d}^+$ in such a way that the point with label $\underline{i}$ has spacial coordinates $(i_1 h_{k,1}, i_2 h_{k,2}, \ldots, i_d h_{k,d})$. Similarly, we label processors in P_k with indices in $s_{n_k,\gamma}^+$.

Thus, processors $\underline{i}$ and $\underline{k}$ are connected if $|\underline{i} - \underline{k}| = 1$. In order to make the machine useful for problems with periodic boundary conditions, we might also add "wrap-around" connections, so that $\underline{i}$ and $\underline{k}$ are connected if $|(\underline{i} - \underline{k}) \bmod n| = 1$. In Section 3.1, it is shown that periodic problems can also be handled without these connections.

Evidently, if each processor has $O(n^{d-\gamma})$ memory cells, we can store the solution, forcing function, and $O(1)$ temporary values belonging to the whole of grid G_k in the processors of P_k; we store gridpoint $\underline{i}$ in processor $\pi_d^\gamma(\underline{i})$ for $\underline{i} \in s_{n,d}^+$.

With the given connectivity, smoothing sweeps of some types can be accomplished in $O(n^{d-\gamma})$

time. It is not necessary for the stencil of the difference scheme to correspond to the connectivity of the processor grid. Jacobi or odd/even SOR smoothing can be so implemented, for example. Let t be the time taken by a single processor to perform the operations at a single gridpoint that, done over the whole grid, constitute a smoothing sweep. If S is the time to implement a smoothing sweep over the whole of grid G_k on processor-grid P_k, then

$$S = t\, n_k^{d-\gamma}. \tag{2}$$

Grid P_k is connected to grid P_{k+1}. Processor $\underline{i} \in P_k$ is connected to processor $a(\underline{i}+\underline{1}) - \underline{1} \in P_{k+1}$. These connections allow the inter-grid operations (forming coarse grid forcing terms b_k and interpolation E_k) to also be computed in O(S) time. We refer to the system of processor-grids $\{P_1, ..., P_J\}$ as the machine L_J for $J = 1, 2, ..., K$.

The execution of the BASICMG iteration by L_k proceeds as follows.

1. First, j smoothing steps on grid G_k are done by P_k. All other processor-grids are idle.
2. The coarse grid equation is formed by P_k and transferred to P_{k+1}.
3. The c cycles of BASICMG on grid k-1 are performed by L_{k-1}.
4. The solution q is transferred to P_k and interpolation $E_k q$ is performed by P_k.
5. The remaining m smoothing steps are done by P_k.

Let W(n) represent the time needed for steps 1, 2, 4 and 5. Then

$$W(n) = (j + m + s)\, t\, n^{d-\gamma} \tag{3}$$

where s is the ratio of the time required to perform steps 2 and 4 to the time needed for one smoothing sweep. Note that s is independent of n, d and γ.

The natural way to build such a machine is to embed the $\gamma = 1$ machine in two dimensions as a system of communicating rows of processors, the $\gamma = 2$ machine in three dimensions as a system of communicating planes, etc. Of course, realizations in three-space are possible for any value of γ. Gannon and Van Rosendale [6] consider the implementation of the fully parallel machine ($\gamma = d$) on proposed VLSI and Multi-Microprocessor system architectures.

This design differs from systolic array designs in that there is no layout with all wire lengths equal. But for reasonably large machines the differences in wire length should not be so great as to cause real difficulties. Moreover, one need not continue to use ever coarser grids until a 1x1

grid is reached. In practice, 3 or 4 levels of grids could be used and most of the multi-grid efficiency retained; this would make the construction much simpler.

3.1. Solving Larger Problems

Suppose there are $(mn)^d$ gridpoints and only n^γ processors. Assume that each processor can store all information associated with $m^d n^{d-\gamma}$ gridpoints. Now we map gridpoints to processors in such a way that neighboring gridpoints reside in neighboring processors. To do this we define a mapping $f_m: s^+_{mn} \rightarrow s^+_n$, such that, for all $i, j \in s^+_{mn}$, $|f_m(i) - f_m(j)| \leq |i - j|$, as follows. Let $j = qn + r$ where q and r are integers, $0 \leq r \leq n-1$. Now let

$$f_m(j) \in \begin{cases} r & \text{if q is even} \\ n-1-r & \text{if q is odd.} \end{cases}$$

Now if m is even, then $f_m(0) = f_m(mn-1) = 0$, so that periodic boundary conditions can be handled without any "wrap-around" connections. This operation corresponds to folding a piece of paper in a fan-like manner; for $m = 10$, for example, like this:

/\/\/\/\/\

To map a multidimensional structure we fold it as above in each coordinate. Let the processor-grid have n^d elements and the point-grid have $m_1 n \times m_2 n \times \ldots \times m_d n$ points. Point $\underline{i}$ can be stored in processor $F_d(m_1, \ldots, m_d, n; \underline{i})$ where $F_d(\underline{i}) = (f_{m_1}(i_1), \ldots, f_{m_d}(i_d))$. If we have only n^γ processors then we map $\underline{i}$ into $F_d^\gamma(\underline{i})$ where $F_d^\gamma(\underline{i}) = F_\gamma(\pi_d^\gamma(\underline{i}))$.

4. Complexity

In this section, we are going to analyse the time complexity of the two multi-grid algorithms, BASICMG and FULLMG, as implemented by the different architectures just discussed. It turns out that the complexity of the two algorithms is very similar. Since the BASICMG algorithm is simpler and is called by FULLMG, we shall discuss and analyse it first. After that, we shall indicate how to derive the results for FULLMG.

4.1. Complexity of BASICMG

To simplify the analysis, we shall assume that the computational domain is a rectangular parallelopiped and is discretized by a hierarchy of cartesian grids (corresponding to the M_j's) each with n_j mesh points on each side (denoted the n_j-grid). Further, we assume that the n_j's satisfy

$n_j = a\,(n_{j-1}+1) - 1$ where a is an integer bigger than one. Generally , we denote by T(n) the time complexity of the BASICMG Algorithm on an n-grid. By inspecting the description of Algorithm BASICMG, it is not difficult to see that T(n) satisfies the following recurrence:

$$T(an) = c\,T(n) + W(an), \tag{4}$$

where W(an) denotes the work needed to preprocess and postprocess the (an)-grid iterate before and after transfer to the coarser n-grid. We have the following general result concerning the solution of (4), the proof of which is elementary.

Lemma 1: *Let T_p be a particular solution of (4), i.e.*

$$T_p(an) = c\,T_p(n) + W(an), \tag{5}$$

then the general solution of (4) is:

$$T(n) = \alpha\, n^{\log_a c} + T_p(n), \quad \text{where } \alpha \text{ is an arbitrary constant.} \tag{6}$$

The term W(an) includes the smoothing sweeps, the computation of the coarse grid correction equation (i.e. the right-hand-side b_{k-1}) and the interpolation back to the fine grid ($E_k q$). The actual time needed depends on the architecture used to implement these operations (specifically the dimensionality of the domain and the number of processors available on an n-grid). In general, as derived in Section 3, W(n) is given by

$$W(n) = (j+m+s)\,t\,g(n) = \beta\, g(n)\,, \tag{7}$$

where $g(n) = n^p$ with $p = d-\gamma$.

In Table (4-1), we give the form of the function g(n) as a function of the architecture and the dimensionality of the domain. We also give a bound on the total number of processors (P) needed to implement the architecture and note that it is always the same order as the number of processors on the finest grid. For a d-dimensional problem with n^γ processors on the n-grid, we have

$$P(\gamma) = \begin{cases} 1 & \text{if } \gamma = 0, \\ (a^\gamma/(a^\gamma-1))\, n^\gamma & \text{if } \gamma > 0. \end{cases}$$

We have the following general result for this class of functions g(n).

Lemma 2: *If $W(n) = \beta\, n^p$, then we can take the following as particular solution of (4):*

$$T_p(n) = \begin{cases} \beta\,(a^p/(a^p-c))\, n^p & \text{if } p \neq \log_a c, \\ \beta\, n^p \log_a n & \text{if } p = \log_a c. \end{cases} \tag{8}$$

Table 4-1: Table of g(n)

Architecture	1-D	2-D	3-D	Total # of processors on all grids, P
1 processor	n	n^2	n^3	1
n processors	1	n	n^2	$(a/(a-1))\ n$
n^2 processors	-	1	n	$(a^2/(a^2-1))\ n^2$
n^3 processors	-	-	1	$(a^3/(a^3-1))\ n^3$

Note: Architecture column gives number of processors on the n-grid.

Combining the results of the last two lemmas, we arrive at our main result:

Theorem 3: The solution of (4) for $W(n) = \beta\, n^p$ satisfies:

$$T(n) = \begin{cases} \beta\,(a^p/(a^p-c))\,n^p + O(n^{\log_a c}) & \text{if } c < a^p, \\ \beta\, n^p \log_a n + O(n^p) & \text{if } c = a^p, \\ O(n^{\log_a c}) & \text{if } c > a^p. \end{cases} \tag{9}$$

Note that in the last case, $\alpha \neq 0$ (in Equation (6)) because $T_p(n)$ does not satisfy the boundary conditions.

For the first two cases ($c \leq a^p$), we can determine the highest order term of T(n) explicitly. However, for the case $c > a^p$, the constant in the highest order term depends on the initial condition of the recurrence (4) (i.e. the time taken by the direct solve on the coarsest grid), which is more difficult to measure in the same units as that of the smoothing and interpolation operations. Fortunately, the complexity for this case is non-optimal and thus not recommended for use in practice and therefore, for our purpose, it is not necessary to determine this constant.

Based on the results in Theorem 3 and the specific forms of the function g(n) in Table 4-1, we can compute the time complexity of Algorithm BASICMG for various combinations of c, a and p, some of which are summarized in Table 4-2, where we tabulated the highest order terms of $T(n)/\beta$.

The classical one processor ($\gamma = 0$) optimal time complexity results [1, 5] are contained in these tables. For example, in two dimensions ($d = 2$) $g(n) = n^2$ and Table 4-2 shows that, for the refinement parameter $a = 2$, $c < 4$ gives an optimal algorithm ($O(n^2)$) whereas $c \geq 4$ is non-

Table 4-2: Time Complexity $T(n)/\beta$ of Algorithm BASICMG

$s = 2$

c	$p = d - \gamma$ = 3	2	1	0
1	$8/7\ n^3$	$4/3\ n^2$	$2\ n$ *	$\log_2 n$
2	$8/6\ n^3$	$4/2\ n^2$ *	$n\log_2 n$	$O(n)$
3	$8/5\ n^3$	$4\ n^2$ *	$O(n^{\log_2 3})$	$O(n^{\log_2 3})$
4	$8/4\ n^3$ *	$n^2\log_2 n$	$O(n^2)$	$O(n^2)$

$s = 3$

c	$p = d - \gamma$ = 3	2	1	0
1	$27/26\ n^3$	$9/8\ n^2$	$3/2\ n$ *	$\log_3 n$
2	$27/25\ n^3$	$9/7\ n^2$	$3\ n$ *	$O(n^{\log_3 2})$
3	$27/24\ n^3$	$9/6\ n^2$ *	$n\log_3 n$	$O(n)$
4	$27/23\ n^3$	$9/5\ n^2$ *	$O(n^{\log_3 4})$	$O(n^{\log_3 4})$

$s = 4$

c	$p = d - \gamma$ = 3	2	1	0
1	$64/63\ n^3$	$16/15\ n^2$	$4/3\ n$ *	$\log_4 n$
2	$64/62\ n^3$	$16/14\ n^2$	$4/2\ n$ *	$O(n^{\log_4 2})$
3	$64/61\ n^3$	$16/13\ n^2$	$4\ n$ *	$O(n^{\log_4 3})$
4	$64/60\ n^3$	$16/12\ n^2$ *	$n\log_4 n$	$O(n)$

s : mesh refinement ratio
c : number of correction cycles in BASICMG
γ : n^7 processors on the n-grid
d : dimension of the domain
***** Asymptotic Efficiency Boundary (See Theorem 5)

optimal. *More generally, we have an optimal scheme if and only if* $c < a^d$. For example, the larger a is or the larger d is, the larger the value c can take for the algorithm to remain optimal. However, with a larger value of a, more relaxation sweeps are needed and a larger value of c has to be taken in order to achieve the same accuracy. We note that the constant in the highest order term of T(n) does not vary a great deal with either c or a (they are all about one, especially for the larger values of a). This suggests that the best balance between speed and accuracy can be achieved by choosing the largest value of c (or close to it) such that the algorithm remains optimal. When $d-\gamma = 2$ and $a = 2$, this means taking c to be 2 or 3.

4.2. Efficiency, Speedup, Accuracy, and Optimal Design

Next, we are going to look at the effects of the new architectures on the performance of the BASICMG algorithm. There are four parameters in this study: c, a, γ and d. We shall call a particular combination of these four parameter a *design*. We shall use the notation $T(c,a,\gamma,d)$ to denote the corresponding complexity of the design. One of the main issues that we would like to address is the *efficiency* E and *speedup* S of a particular design, which are defined as [8]:

Definition 4:

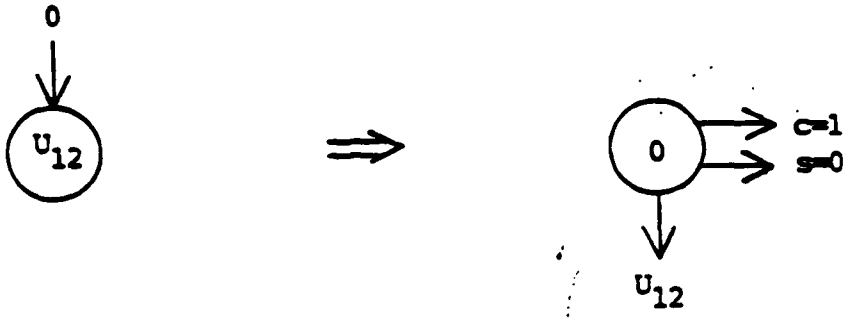
$$S(c,a,\gamma,d) = T(c,a,0,d) \,/\, T(c,a,\gamma,d)\,,$$

$$E(c,a,\gamma,d) = T(c,a,0,d) \,/\, (\, P(\gamma)\, T(c,a,\gamma,d)\,)\,.$$

The speedup S measures the gain in speed over the one processor architecture while the efficiency E reflects the tradeoff between processors and time and measures the efficiency with which the architecture exploits the extra processors to achieve the speedup. The optimal efficiency is unity, in which case a P-fold increase in the number of processors reduces the time complexity P-fold. In general, the efficiency E and the speedup S are functions of n. We shall call a design *asymptotically efficient* if E tends to a constant as n tends to infinity and *asymptotically inefficient* if it tends to zero. We shall primarily be concerned with analysing the efficiency E of a design in this section. The speedup S can be easily read from Table 4-2.

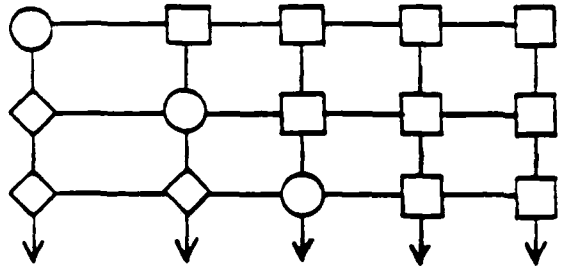
For determining the asymptotic efficiency of a design, it suffices to determine the highest order term of E. The efficiency E can be derived from the explicit expressions for T and P in a straightforward manner. Since the efficiency for $\gamma = 0$ is unity by definition, we shall only be interested in $\gamma > 0$. We summarize the results in the following theorem.

load $u_{1,k}$ in, for $k = 3,4,\ldots,q$. At time 4, this special function is required at cell (2,2) again:



This rotation follows the earlier one down the row, removing elements $u_{1,k}$ and loading zeros. In general, cell (k,k) performs the special function k times, at $t = 2k - 1,\ldots,3k - 2$. Then k identity rotations flow down row k, pushing out rows $k, k - 1,\ldots,1$ of U, and finally loading the zero that precedes the next matrix.

To make the array output uniform, we would add some cells at the lower left to make the array a rectangle:



The only purpose of these diamond-shaped cells is to delay final output of elements of U, which now leave the array in the same format as elements of A – element i,j leaves at relative time $m - i + j$.

3. The Backsolve Array

To solve the triangular system $U^T Y = B$ (Y and B are $m \times n$) we can use the triangular array shown in Figure 6. The details of this array are straightforward and are omitted. We note that it consists of a triangular array of cells each containing a single element of U exactly as does the GK array, so that it might in some applications be useful to build a common realization of both these arrays.

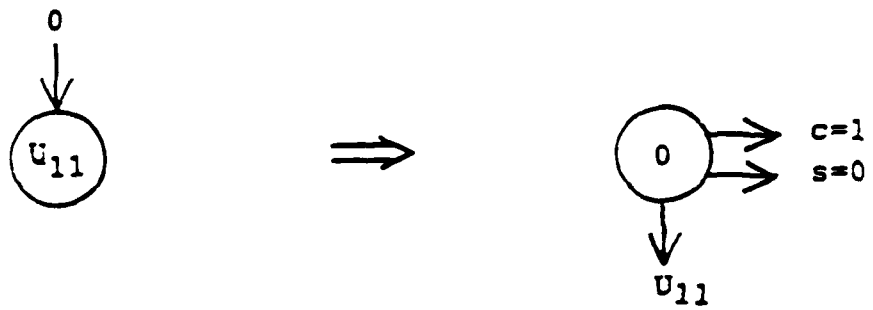
In the present context, n is often large. We intend to solve two systems, $U^T Y = B$, then $UX = Y$; we are not otherwise interested in elements of Y. If X is stored we can store Y in its place. Suppose, however, that X will not be stored. We may want to minimize temporary storage for Y. This can be reduced to $O(m^2)$ locations in two ways. We could use a second array to solve $UX = Y$ and stream the first array's output into this second array. An interface of $3m(m - 1) / 2$ delay cells is needed, as Figure 7 shows. There is another possibility. One array can solve both systems at the same time. Figure 8 shows two successive cycles of such a device. At a given instant, every second diagonal is working

so the two matrices are separated by a line of zeros. The scheme will, in effect, push out U as it is created and fill each cell with a zero just before a B - element reaches it. It therefore "looks" to the new matrix B as if the array initially contained only zeros.

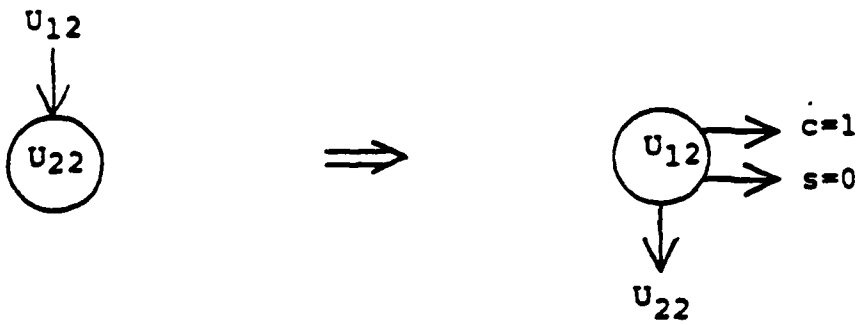
When a B - element first arrives at a boundary cell it meets a zero (we shall later show this). The boundary cell's normal function (see Figure 2) is to store this element's absolute value in its memory and output the identity rotation, if the element was non-negative, or - 1 times the identity rotation otherwise. As this rotation moves to the right it meets cells containing zeros in their memories and pushes these zeros out, loading instead elements (possibly negated) of row n of B. Thus the zeros continue to lead the columns of B down through the array.

We now show how elements of U are unloaded. Let time $t = 0$ be the time a_{11} enters cell (1, 1). Then cell (i,j) accepts its last A - element, and computes its U - element, thereby finishing its work, at time $t = i + j - 2$. By our assumed sequence of inputs (4) the datum zero appears at the input to cell $(1,j)$ at time j, immediately after it has computed $u_{1,j}$. To make the scheme work, we want an identity rotation to get there at the same time, knocking out the computed element $u_{1,j}$ and loading the zero. This will be made to happen by a special boundary cell function.

Let cell (1,1) do this at time 1:



The rotation so generated will reach cell $(1,j)$ at time j, as required. Now, at time 3, let cell (2,2) do the same thing:



(The datum $u_{1,2}$ has been forced out of the first row, as described above). This rotation will move to the right and knock elements $u_{2,k}$ out of cells $(2,k)$ and

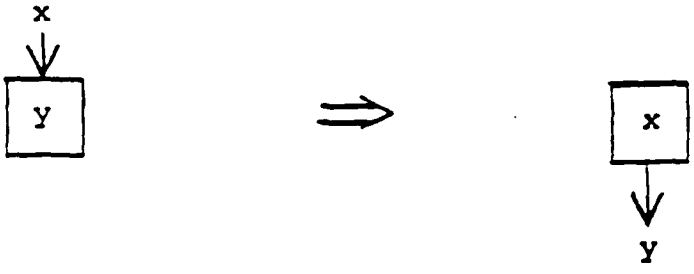
1) / 2, of the array. What is the best shape possible? Two conflicting factors influence the decision. The I/O bandwidth to support the array is least for a "well-rounded" array ($p \approx q$) since only the cells at the array edge communicate with the surrounding systems. But the number of passes needed to solve the given problem will usually not be minimized by taking $p = q$. In typical cases (say $m = 100$, $pq - p(p-1)/2 = 31$) the minimizing shape may be quite narrow ($p = 2$, $q = 16$ in this example).

2.3. Unloading the Cell Memories

The *QU* trapezoid implements the matrix factorization (3). But how can we remove the elements of U_{11} and U_{12}, which are stored in the cells of the array? Here we shall develop a scheme with these properties:

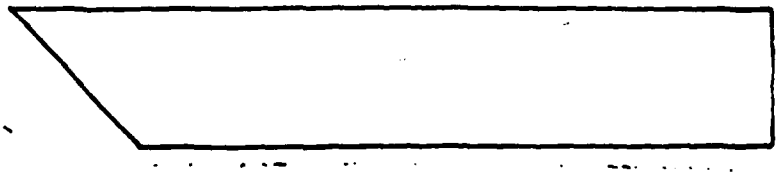
1) The outward flow of data is entirely uniform.
2) Control signals are applied only at the boundary cells.
3) No specially controlled functions are required of the internal cells.
4) The array can finish the factorization of a matrix, unload its cell memories, and begin the factorization of another matrix with no delay whatever.

The key to the unloading scheme is the way an internal cell behaves when given the "identity" rotation ($c = 1$, $s = 0$). It acts as a unit-delay:



To begin, suppose a new matrix B follows the input matrix A. The data will be presented to the array in this format,

$$
\begin{array}{ccccccc}
 & & & & & & b_{nm} \\
 & & & & & & \cdot \\
 & & & & & & 0 \\
 & & & & & & \cdot \\
 & & b_{n2} & & \cdot & & a_{1m} \\
 & & \cdot & & & & \cdot \\
b_{n1} & & 0 & & \cdot & & \\
 & & \cdot & & & & \\
0 & & a_{12} & & & & \\
 & & & & & & \\
a_{11} & & a_{22} & \cdot & \cdot & \cdot & a_{mm}
\end{array}
\qquad (4)
$$

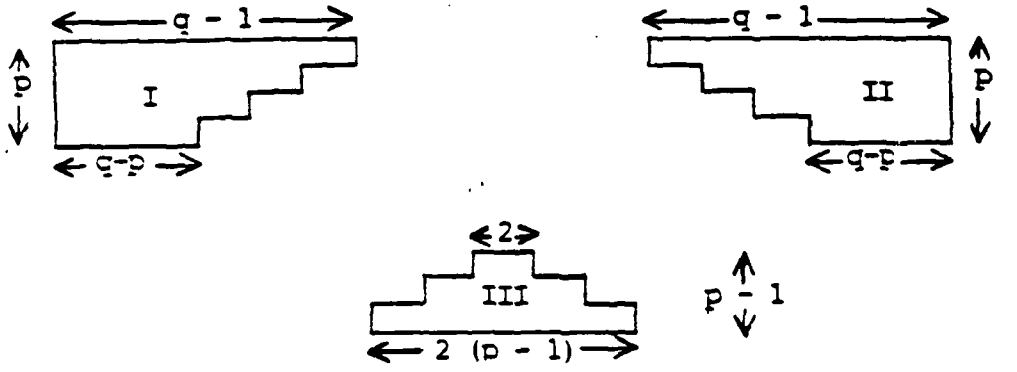


stored in the bank above the cell it enters first. A column (of temporary results) that emerges from the bottom is sent to the bank above the cell it will next enter. When the passes are sequenced as in Figure 4, this destination is for some columns uniquely determined, and for others is one of two possibilities. Thus, the extra-array interconnections are very simple.

Control of the memories is also simple, since the pattern of access to the data, (Figure 7) is so regular. Memory addresses could be generated once and passed from one bank to the next.

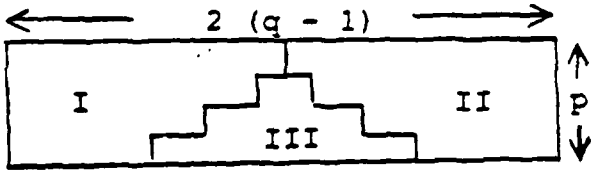
2.2.1. An alternate scheme

There is another possibility. We could also work with tiles of these shapes,



The array implements tile II by using all internal cells actively, tile I in the same way, but with columns brought in reverse order, and tile III by shutting off certain internal cells. Tile III can be realized only if it fits into the array – if $2(p-1) \leq q-p$.

Comparing the two possibilities we see that the first has an average tile size of $p(q-p)$ while the second has $2p(q-1)/3$ (the tiles are used to cover rectangles like this:)



Thus, the first scheme is more efficient if $p(q-p) > 2p(q-1)/3$, i.e. if $q > 3p-2$. Since the second scheme *requires* that q be at least $3p-2$, it can never be more efficient than the first.

2.2.2. Choosing the Array Shape

We suppose that some constraint, cost for example, limits the size, $pq - p(p-$

compute the factorization

$$Q^*B = \begin{bmatrix} U_{11} & U_{12} \\ O & B_2 \end{bmatrix} \tag{3}$$

where O denotes the zero matrix order $(n-p)\times p$, U_{11} is $p\times p$ upper triangular, and U_{12} and B_2 are full $(q-p)$ - column matrices. B_2 emerges from the bottom of the array; U_{11} and U_{12} are stored in it.

Can we obtain a complete QU factorization with the trapezoid? If q is not less than m, the number of columns of A, we can; we would zero A in groups of p columns, from left to right, using $\lceil m/p \rceil$ passes through the array. The details are obvious. But if $m > q$ we cannot. For suppose we zero the first p columns of A below the diagonal by passing columns $1,2,\ldots,q$ through the array. We want next to zero columns $p+1,\ldots,2p$, by passing $p+1,\ldots,p+q$ through. But until we have applied the rotations from the first pass to columns $q+1,\ldots,q+p$, we may not apply those of the second pass.

To allow factorization of matrices with more than q columns, the array must provide a second function: the ability to apply previously computed (and stored) rotations to a set of input columns. We can give the array this ability by turning off the cells in the leftmost $p\times p$ triangular part, keeping a $p\times(q-p)$ rectangle active. We also allow rotation parameters to come in via the left edge. A set of $q-p$ columns can enter the active rectangle at the top. The result – the input rotations applied to the input columns – emerges from the bottom, except for the first p rows, which are stored in the cells of the active rectangle.

2.2. Simulating The Full Array

Here we show how the $p\times q$ trapezoidal array, supported by an appropriate memory system for partial results, can generate a QU factorization when $m > q$. Imagine that the set of work to be done is represented by an $m\times m$ triangular array of pairs,

$$(i,j) : 1 \le i \le j \le m$$

where the pair (i,j) represents the task of applying to column j the rotations used to zero elements of column i. The array can be used to perform "generate" passes, where columns are actually zeroed, and "apply" passes where stored rotations are applied. A generate pass performs a $p\times q$ trapezoidal piece of the set of task pairs; an apply pass performs a $p\times(q-p)$ rectangular piece. Sequencing the passes to perform the entire job is analogous to covering a triangle by trapezoidal and rectangular "tiles" following these rules:

(R1) Trapezoidal tiles must be placed at the triangle's diagonal edge;
(R2) No tile may be placed unless the diagonal edge to its left has been tiled;
(R3) No tile may be placed if any space directly below it is untiled.

There are many legal tilings; Figure 4 shows one.

In generating the QU factorization by multiple passes, temporary results are produced. These must be stored and reentered into the array later. Figure 5 shows a suitable memory design. The important features are these. There is a separate, independent memory bank for each array column. A matrix column is

directions: complex matrices, unloading the result, U, from the array, control and synchronization, fabrication of the cells, and simulation of a large array by a physically smaller array through decomposition of the problem.

QU factorization of A is performed by finding an orthogonal matrix, Q^*, such that $Q^*A = U$ is upper-triangular. Q^* can be a product of simple orthogonal matrices (Givens rotations) each chosen to zero one element of A below the diagonal. To zero $a_{i,j}$, the i^{th} and $(i-1)^{th}$ rows of A are replaced by

$$\begin{bmatrix} a_{i-1,k} \\ a_{i,k} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_{i-1,k} \\ a_{i,k} \end{bmatrix}, \qquad k = 1, 2, \ldots, m$$

where (c,s) are chosen so that $a_{i,j}$ becomes zero and the matrix shown is orthogonal:

$$c = \frac{a_{i-1,j}}{\sqrt{a_{i-1,j}^2 + a_{i,j}^2}}$$

$$s = \frac{a_{i,j}}{\sqrt{a_{i-1,j}^2 + a_{i,j}^2}}$$

The elements can be zeroed column by column from the bottom up; elements are zeroed in the sequence

$$(n, 1), (n-1, 1), \ldots, (2, 1); (n, 2), \ldots, (3, 2); \cdots; (n, m), \ldots, (m+1, m)$$

Note that the rotation zeroing $a_{i,j}$ needs to be applied only to columns $j, j+1, \ldots, m$ since, at the time it is applied, the elements $a_{i-1,k}$ and $a_{i,k}$ for $k < j$ are already zero.

The Gentleman-Kung (GK) array, an $m \times m$ triangular array of two cell types that computes the QU factorization (1) of an $n \times m$ input matrix A, is shown in Figure 1. We call the circles "boundary" cells. Figure 2 shows the cell's functions. We shall now explain the GK array's operation. First, note that the cells each have a single memory. These initially are zero. The diagonal boundary cells compute rotation parameters (c,s). Cell (j,j) computes the rotations that zero elements of column j of A. These rotations then move right, along the rows of the array. The square "internal" cells apply these rotations to the other columns.

The matrix A enters the array at the top in the pattern shown in Figure 1. To the upper left of each cell is the time that the first element of A arrives. Suppose $a_{n,1}$ enters cell (1,1) at time $t = 1$. The first element of U, $u_{1,1}$, is computed in cell (1,1) at time $t = n$. By time $t = n + 2(m-1)$ the last element, $u_{m,m}$, has been computed. U now resides in the array. The rotations defining Q^* will have emerged from the right edge.

2.1. The Trapezoidal Subarray

We would like to solve beamforming problems of various sizes using one physical array, so we must consider how to simulate a full $m \times m$ array using a smaller piece. Suppose we have a $p \times q$ trapezoid, as shown in Figure 3. Let B be a matrix having n rows and q columns. If B is presented at the array top, we

$$w = \frac{R_{xx}^{-1} c}{c^* R_{xx}^{-1} c}$$

where c is the desired signal vector, and R_{xx} is the covariance matrix of the signal at frequency ω:

$$R_{xx}(\omega) = E\{x(\omega)x^*(\omega)\}$$

In practice, for every interesting frequency, an $n \times m$ matrix of samples of the signal

$$X^*(\omega) = \begin{bmatrix} {}^1x^*(\omega) \\ {}^2x^*(\omega) \\ \vdots \\ {}^nx^*(\omega) \end{bmatrix}$$

would be obtained, and R estimated by

$$R \approx XX^*.$$

(Here, * denotes conjugate transpose). Possibly, different weights would be given to the rows of $X^*(\omega)$.

The following algorithm gives the solution.

1. Factor

$$X^* = Q\,U \tag{1}$$

where Q is an $n \times n$ unitary matrix, and U is the $n \times m$ matrix

$$U = \begin{bmatrix} U \\ O \end{bmatrix}$$

2. For each bearing-angle ϑ,

(a) forward solve:

$$U^* a(\vartheta) = c(\vartheta) \tag{2a}$$

(b) back solve:

$$U\,w(\vartheta) = a(\vartheta)\,(a^* a)^{-1} \tag{2b}$$

(Here $a = (U^*)^{-1} c$, so $a^* a = c^* U^{-1}(U^*)^{-1} c = c^* R_{xx}^{-1} c$)

For the remainder of this paper we shall concentrate on the design of an adaptive weight-selection processor that performs the two major steps of the algorithm. Two systolic arrays will be used. One, a variant of the design of Gentlemen and Kung for QU factorizations [2], performs step 1. The second does the triangular solves of step 2 and is new.

2. The QU - factorization Processor

This section is an extension of previous results of Gentlemen and Kung in several

Systolic Linear Algebra Machines In Digital Signal Processing

Robert Schreiber *

Philip J. Kuekes
ESL, Incorporated
Sunnyvale, CA 94086

1. Introduction

Several recent contributions to the literature in signal processing, computer architecture, and VLSI design showed that systolic arrays are extremely useful for designing special purpose high-performance devices to solve problems in numerical linear algebra [1,2,3,4,5]. But no attention has been paid to the problems of integrating these designs into any computing or signal processing environment. The purpose of this paper is to examine systolic arrays in a specific context. We have chosen an adaptive beamforming problem as that context.

The adaptive beamforming problem chosen is simple, yet typical of those encountered in sonar signal processing applications. The signals of a large array of m identical sensors are sampled, stored and Fourier transformed in time. The result is a set $x(\omega,i)$ of complex values depending on frequency (ω) and sensor (i). Then, for each resulting frequency ω an array output function $g(\omega,\vartheta)$, depending on a bearing-angle ϑ, is produced, by

$$g(\omega,\vartheta) = \sum_{i=1}^{m} x(\omega,i)\,\overline{w}(i,\omega,\vartheta)$$

Here overbar denotes complex conjugate. The vector w determines the characteristics of the beamformer. For the minimum-variance distortionless response beamformer, w is chosen to minimize the output power, the expected value of $|g|^2$, subject to a signal-protection constraint

$$\sum_{i=1}^{m} c(i,\omega,\vartheta)\overline{w}(i,\omega,\vartheta) = 1$$

Here $c(i,\omega,\vartheta)$ is the output of sensor i at frequency ω given no signal other than that coming from a source at bearing-angle ϑ.

The solution is to choose the weight vector

$$w(\omega,\vartheta) = (w(1,\omega,\vartheta), \ldots, w(m,\omega,\vartheta))^T$$

as

---

* Permanent address: Department of Computer Science, Stanford University, Stanford, CA 94305

References

[1] R. E. Bank and T. Dupont.
An optimal order process for solving elliptic finite element equations.
Math. Comp. 36:35-51, 1981.

[2] A. Brandt.
Multi-level adaptive solutions to boundary-value problems.
Math. Comp. 31:333-390, 1977.

[3] A. Brandt.
Multi-Grid Solvers on Parallel Computers.
Technical Report 80-23, ICASE, NASA Langley Research Center, Hampton, VA, 1980.

[4] P. G. Ciarlet.
The Finite Element Method for Elliptic Problems.
North-Holland, Amsterdam, 1978.

[5] C. Douglas.
Multi-Grid Algorithms for Elliptic Boundary-Value Problems.
PhD thesis, Yale University, 1982.
Computer Science Department Technical Report 223.

[6] D. Gannon and J. van Rosendale.
Highly Parallel Multi-Grid Solvers for Elliptic PDEs: An Experimental Analysis.
Technical Report 82-36, ICASE, NASA Langley Research Center, Hampton, VA, 1982.

[7] W. Hackbusch.
A Multi-Grid Method Applied to a Boundary Value Problem with Variable Coefficients in a Rectangle.
Technical Report 77-17, Mathematisches Institut, Universitat zu Koln, Cologne, 1977.

[8] David J. Kuck.
The Structure of Computers and Computations.
John Wiley & Sons, New York, 1978.

[9] H.T. Kung.
Why Systolic Architectures?
IEEE Computer 15(1):37-46, January, 1982.

unity for the values of a and p that occur. The r part of the constant applies equally to all entries of Table 4-2 and reflects the number of times BASICMG is called by FULLMG. We point out that the choice of r that results in truncation error level accuracy depends on how efficiently BASICMG reduces its initial error but can be chosen *independent of n* [5]. Thus, the extra multiplicative factor does not affect the asymptotic efficiency of a particular entry in Table 4-2. The complexity of F(n) in the last case is actually increased by a factor of $\log_a n$ over that of T(n). However, the corresponding entries in Table 4-2 are already asymptotically inefficient and thus this extra factor again does not affect the asymptotic efficiency of the design. It follows that the discussions concerning the efficiency, speedup and optimal design for the BASICMG algorithm in Section 4.1 are also valid for the FULLMG algorithm, with the exception that a fully parallel logarithmically asymtotically efficient design is slower by the factor $\log_a n$.

5. Conclusion

We have proposed an architecture based on a system of processor-grids for parallel execution of multi-grid methods based on a system of point-grids. We have analyzed its efficiency and shown that a combination of algorithm and machine is asymptotically efficient if and only if $c < a^{d-\gamma}$, where

- c is the number of coarse grid iterations per fine grid iterations,
- a is the mesh refinement factor,
- d is the dimension of the point-grids,
- γ is the dimension of the processor-grids.

We find therefore that fully parallel designs — with $\gamma = d$ — cannot be asymptotically efficient. There is, however, only a logarithmic fall-off in efficiency when $c = a^{d-\gamma}$, and for fully parallel designs this occurs for $c = 1$.

4.3. Complexity of FULLMG

In this section, we shall derive the complexity of the FULLMG Algorithm. Since FULLMG calls BASICMG, the results here depend crucially on the complexity of Algorithm BASICMG.

Let F(n) denote the time taken by one call to FULLMG. By inspecting the algorithm in Table 2-2, it can easily be verified that F(n) satisfies the following recurrence:

$$F(an) = F(n) + r\,T(an)\,, \tag{10}$$

where we have absorbed the cost of the interpolation step in FULLMG into the interpolation costs of BASICMG (i.e. the term s in Equation (7)). Note that this is just a special case of the recurrence (4), with c = 1 and W(an) = r T(an). By inspecting the entries in Table 4-2, we see that the forcing function W(n) in this case takes the form of either n^p or $n^p\log_a n$. We have the following result for particular solutions of (10) for this class of forcing functions, which can be verified by direct substitution.

Lemma 6:

(1) If $T(n) = \alpha n^p$ then a particular solution of (10) is:

$$F_p(n) = (a^p/(a^p-1))\,r\alpha n^p\,.$$

(2) If $T(n) = \alpha n^p\log_a n$, with $p > 0$, then a particular solution of (10) is:

$$F_p(n) = (a^p/(a^p-1))\,r\alpha n^p\log_a n - (a^p/(a^p-1)^2)\,r\alpha n^p\,.$$

(3) If $T(n) = \alpha\log_a n$ then a particular solution of (10) is:

$$F_p(n) = (r\alpha/2)\,(\log_a^2 n + \log_a n)\,.$$

Since the homogeneous solution of (10) is a constant, the general solution is dominated by the particular solutions. We give the highest order terms of F(n) in terms of T(n) in the following theorem.

Theorem 7:

(1) If $T(n) = \alpha n^p$ then $F(n) = (a^p/(a^p-1))\,rT(n) + O(1)$.

(2) If $T(n) = \alpha n^p\log_a n$, with $p > 0$, then $F(n) = (a^p/(a^p-1))\,rT(n) + O(n^p)$.

(3) If $T(n) = \alpha\log_a n$ then $F(n) = (r/2)\log_a n\,T(n) + O(\log_a n)$.

In the first two cases, the complexity of F(n) is the same as that of T(n), except for the *constant* multiplicative factor $(a^p/(a^p-1))r$. The $(a^p/(a^p-1))$ part of this constant is very close to

Next we shall fix a and γ and vary c (columns in Table 4-2). That is, we fix the architecture and vary the multi-grid algorithm. This time the speedup factor S decreases slightly as we increase c which is not surprising as we are doing more work on coarse grids, and this keeps many processors idle. Again, the efficiency E decreases as we increase c, and after a certain entry, the algorithm becomes asymptotically inefficient. For example, take the two dimensional case with n processors on the n-grid ($d = 2$, $\gamma = 1$) and $a = 3$. Going down the appropriate column, we have $E(1,3,1,2) = = 1/2$, $E(2,3,1,2) = 2/7$ and $E(3,3,1,2) = 1/\log_3 n$. Recall that the larger c is, the more accurate is the computed solution and the more robust is the overall algorithm. Thus, the $c = 2$ design is the most accurate efficient design. *In general, for fixed a and γ, the design just above the efficiency boundary is the most accurate efficient design. If accuracy is no problem, then c can be chosen smaller to speed up the algorithm.*

Finally, we fix c and γ and vary a. Generally, a larger value of a means fewer processors are needed to implement the architecture. It also means that less work has to be done on the coarse grids because they have fewer points. To see the effect of varying a, note that the efficiency boundary moves towards the lower right hand corner of the tables in Table 4-2 as a is increased. This implies that, for a fix architecture (γ) and algorithm (c), using a larger value of a will generally exploit the available processors more efficiently. However, one cannot indiscrimatorily use large values for a because this leads to larger interpolation errors and less accurate solutions. For example, take the two dimensional case with n-processors on the n-grid and $c = 2$. With $a = 2$, the algorithm is asymptotically inefficient ($E(2,2,1,2) = 1/\log_2 n$) whereas with $a = 3$ and $a = 4$, it is asymptotically efficient ($E(2,3,1,2) = 2/7$, $E(2,4,1,2) = 3/7$). Thus, the $a = 3$ design is the most accurate efficient design. *In general, for fixed c and γ, the smallest value of a that yields an efficient design is the most accurate efficient design. If accuracy is no problem, then a can be chosen larger to speed up the algorithm.*

One can carry out similar parametric studies, for example, by fixing one parameter and varying the other two. For instance, if we are free to choose both the architecture (γ) and the algorithm (c), then by inspecting the form of the efficiency constraint $c < a^{d-\gamma}$, it can be seen that smaller values of c allow more processors to be used (larger γ) to produce a faster efficient design. For example, in the $a = 2$ case, if $c = 2$ then the $p = 2$ entry gives the fastest efficient design whereas if $c = 1$, it becomes the $p = 1$ entry, with the latter being faster. Similarly, for a fixed c, a larger value of a allows more processors to be used to achieve a faster efficient design.

Table 4-3: Influence of Design Parameters on Optimality Conditions

		Design Parameters		
		c	a	γ
Optimality Conditions	Max Accuracy	large	small	indep
	Min T(n)	small	large	large
Constraint	Efficiency E	small	large	small

In Table 4-3, we indicate the influence of each of the three design parameters $\{c,a,\gamma\}$ on the optimality conditions and the constraint. For example, the accuracy of the solution is independent of γ and to achieve maximum accuracy, one should take c large and a small. The appropriate choice of optimality condition depends on the requirements of the given problem. Moreover, the general optimal design problem may not have a unique or bounded solution in the three parameter space $\{c,a,\gamma\}$. In practice, however, we usually do not have the freedom to choose all three parameters. If the number of free parameters are restricted, then the optimal design problem may have a unique solution.

We shall illustrate this by fixing two of the three parameters in turn and study E as a function of the free parameter. First, let us fix c and a and consider the effect of varying γ. In other words, we consider the case where the multi-grid algorithm and the refinement of the domain are fixed and we are free to choose the architecture. Varying γ corresponds to moving across a particular row of Table 4-2. It is easy to see that one achieves a speedup as we use more processors (i.e. as one moves from left to right in one of these rows). However, the efficiency E generally goes down as one uses more processors, and after a certain entry the design starts to be asymptotically inefficient. For example, take the three dimensional case ($d = 3$), with $a = 2$ and $c = 2$. With n processors on the n-grid, the efficiency is $E(2,2,1,3) = 1/3$. With n^2 processors on n-grid, we have $E(2,2,2,3) = 1/\log_2 n$, and with n^3 processors $E(2,2,3,3) = O(1/n)$. Both the last two designs are asymptotically inefficient and thus the $\gamma = 1$ design is the fastest efficient design. ***In general, for fixed c and a, the design just to the left of the efficiency boundary is the fastest efficient design.***

Theorem 5: Assume $\gamma > 0$.

(1) If $c < a^{d-\gamma}$ then $E(c,a,\gamma,d) = (a^{\gamma}-1)(a^{d-\gamma}-c)/(a^{d}-c)$.

(2) If $c = a^{d-\gamma}$ then $E(c,a,\gamma,d) = (a^{\gamma}-1)a^{d-\gamma} / ((a^{d}-c)\log_a n)$.

(3) If $c > a^{d-\gamma}$ then

$$E(c,a,\gamma,d) = \begin{cases} O(1/(n^{\log_a c-d+\gamma})) & \text{if } c < a^{d}, \\ O(\log_a n / n^{\gamma}) & \text{if } c = a^{d}, \\ O(1/n^{\gamma}) & \text{if } c > a^{d}. \end{cases}$$

Based on the above theorem, we can immediately make the following observations:

1. *A design is asymptotically efficient if and only if* $c < a^{d-\gamma}$. This inequality defines an *efficiency boundary* in the four parameter space of $\{c,a,\gamma,d\}$, the projections of which are shown by *'s in Table 4-2.
2. *The fully parallel design ($\gamma = d$) is always asymptotically inefficient.* This follows because to have an efficient design in this case requires $c < 1$ which is meaningless for the multi-grid algorithm.
3. Define a logarithmically asympototically efficient design to be one with $E = O(1/\log_a n)$ as n tends to infinity. *A fully parallel design is logarithmically asymptotically efficient if and only if $c = 1$.* This is case (2) in Theorem 5.
4. *If we start with a non-optimal design in the one processor case, then adding more processors will not make the design asymptotically efficient.* This corresponds to the last two cases in Case (3) of Theorem 5. The reason is that too many coarse grid correction cycles are performed so that even if more processors are added to speed up the setup time for transferring to the coaser grids, too much time is spent on the coaser grids.

Asymptotically efficient designs are theoretically appealing. They indicate that the extra processors are utilized efficiently to achieve the speedup. For this reason, it is interesting to consider the following problem:

Optimal Design Problem:
For a given problem (i.e. given d), find the design that
minimizes T(n) and/or maximizes the accuracy of the computed solution
subject to the constraint that it is asymptotically efficient.

on one problem, the other diagonals on the second problem. An array of $(3m^2 - 2m) / 4$ delay cells is needed; about half as many as in the 2 - array design.

3.1. Computing The Scale Factors

The computation (2b) requires the scale factor $a^*(\vartheta)a(\vartheta)$, that is the dot product of the solution of the system (2a) with itself. Since the solution vectors a are produced one element per cycle by successive array columns, we can accumulate these dot products by attaching a row of cells at the bottom of the array; see Figure 6, for example.

4. Complex QU Factorization

For signal processing applications, we must deal with complex matrices A. Of course, one can solve a complex $n \times m$ least squares problem by means of a real QU factorization of the $2n \times 2m$ matrix

$$\begin{bmatrix} A_R & -A_I \\ A_I & A_R \end{bmatrix}$$

where $A = A_R + iA_I$ is the decomposition of A into its real and imaginary parts. But when Givens rotations are used, this factorization requires $8/3$ times as many real multiplications as a direct complex QU factorization of A. We shall now discuss how this can be done.

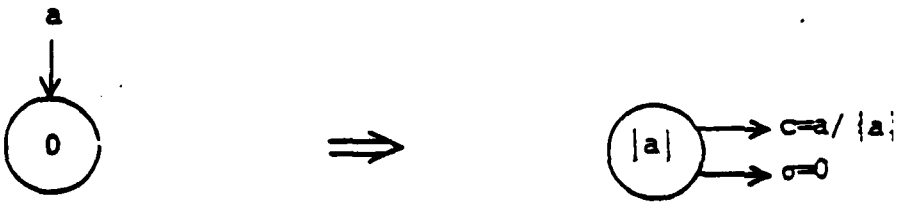
The QU factorization is unique only up to scaling of the rows of U by factors of unit modulus: $A = (QD)\ (D^{-1}U)$ is also a QU factorization for any unitary diagonal matrix D. Thus, we may require that the diagonal elements of U be positive real. This is the (unique) factorization computed by our array.

Let us change the cell definitions of Figure 1 to those of Figure 9. (We shall now employ this convention: lower case Roman letters are complex, Greek are real, and, for example,

$$a = \alpha + j\alpha' \qquad x = \xi + j\xi'$$

where $j = \sqrt{-1}$. These cells are, of course, more difficult to implement than the real Givens cells.

With this there is little else that changes. Now, when a leading element of an input matrix hits a boundary cell the effect is this

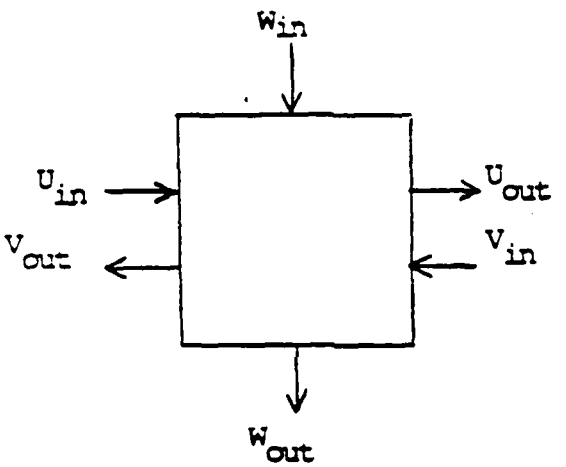


Thus, instead of the identity produced in the real case, a unitary diagonal rotation that simply rescales a row is produced.

6. VLSI Implementation

In this section we shall discuss how the internal cells of the real and complex *QU* arrays and the complex backsolve array might be fabricated using a Systolic Internal Chip (SIC) that is being developed at ESL. It is particularly noteworthy that these different cells can be obtained using a common VLSI building block, without the use of additional chips. Moreover, we have used the SIC to design other compound cells (for real and complex *LU* factorization of dense and band matrices and for band *QU* factorization). We expect that systolic arrays for most of the standard algorithms of numerical linear algebra can be generated using this chip and one other that we will also mention.

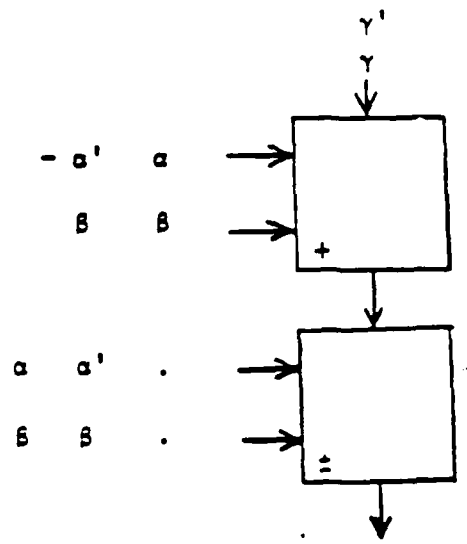
First we shall give a rough description of the SIC. It is being designed in a TRW 2μ CMOS technology. Its function is this



$$U_{out} = U_{in}$$
$$V_{out} = V_{in}$$
$$W_{out} = W_{in} \textcircled{s} U_{in} V_{in}$$

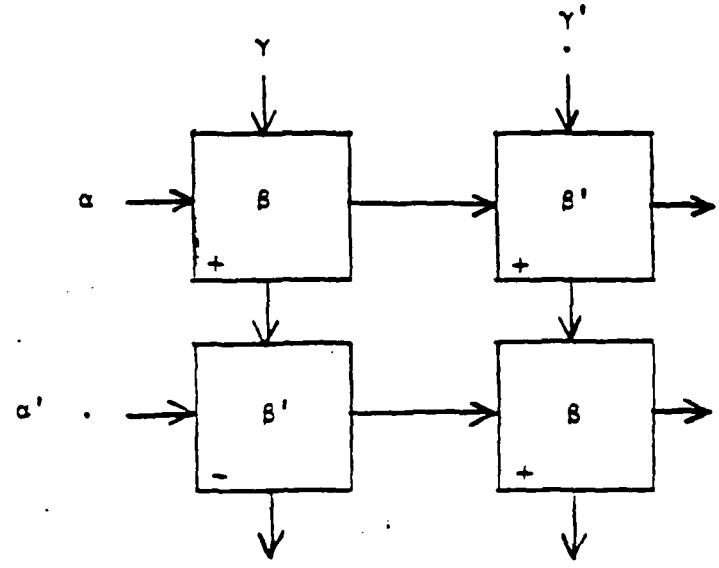
where $\textcircled{s} \in \{+, -, \pm, +-\}$ denotes the sign used in the addition. This can be $+$, $-$, or can alternate between the two. Operands are floating point. Substantial effort has gone into providing switching functions and internal registers to increase the SIC's flexibility.

Compound cells for complex arithmetic can be built. Here are two of Kung's designs: a complex $c + ab$ cell using 2 SICs.



$$(c \equiv \gamma + j\,\gamma',$$
$$a \equiv \alpha + j\,\alpha',$$
$$b \equiv \beta + j\,\beta'.)$$

and another, using 4 chips, in which one operand is stationary.



This is the internal cell for the complex backsolve array.

Figure 10 gives a layout for a complex Givens cell that uses 6 SICs. There are two two-chip complex multiply-add cells, one for c^*x and another for cy, and two one-chip cells for real * complex multiply-add, one for σx and another for σy. The complex quantities are represented using one of the formats discussed above. New operands can enter the cell every second clock. There is a three clock delay on the $x-z$ path, but only a 1 clock delay on the $c_{in}-c_{out}$ and $\sigma_{in}-\sigma_{out}$ paths.

We lack the space to fully discuss the boundary cell's implementation. Another chip is necessary. A chip using either faster gates or more internal parallelism could provide divide, square root, and reciprocal square root operations at a rate of one operation per SIC cycle. This second chip could, with the SIC, be used to design a pipelined boundary cell for *QU*, backsolve, or *LU* operations with enough throughput to keep pace with the array. The boundary cells will usually have more latency than the internal cells.

For a trapezoidal *QU* array this extra boundary cell latency throws the array out of synchronization unless we provide appropriate intercell delays. Fortunately these delays are needed only in the left-hand triangular part of the array. Figure 11 gives an example. Here the boundary cell's latency is 5 and the internal cell's is 4, although its left-right latency is just 1. The time an operand first arrives is shown in each cell. The diamonds are delays; they delay the data 4 cycles. Their effect is to equalize the latency of alternate paths through the array. A $p \times q$ array requires $p(p-1)/2$ delay cells. These delays have an effect on the latency of the array and on the pattern of input and output (it is not a parallelogram any more). But there is no reduction in throughput.

ACKNOWLEDGMENT

We gladly thank Dr. Theo Kooij of DARPA for encouraging this work and suggesting and explaining to us the beamforming problem. We also thank our colleagues at ESL, Geoffrey Frank, Dragan Milojkovic, and Larry Ruane for their helpful comments.

REFERENCES

1. A. Bojanczyk, R.P. Brent, and H.T. Kung, "Numerically Stable Solution of Dense Systems of Linear Equations Using Mesh-connect Processors", Technical report CMU-CS-81-119, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1981.

2. W.M. Gentleman and H.T. Kung, "Matrix Triangularization by Systolic Arrays", *Real Time Signal Processing IV*, SPIE Vol. 298, Society of Photo-Optical Instrumentation Engineers, Bellingham, Wa., 1981.

3. Don E. Heller and Ilse C. F. Ipsen, "Systolic Networks for Orthogonal Decompositions with Applications", Tech. Rept. CS-81-18, Computer Science Dept., Pennsylvania State University, University Park, Pa., 1981.

4. Robert Schreiber, "Systolic Arrays for Eigenvalue Composition", *Real Time Signal Processing V*, SPIE Vol. 341, Society of Photo-Optical Instrumentation Engineers, Bellingham, Wa., 1982.

5. H. J. Whitehouse and J. M. Speiser, "Sonar Applications of Systolic Array Technology", IEEE Eascon Proceedings, 1981.

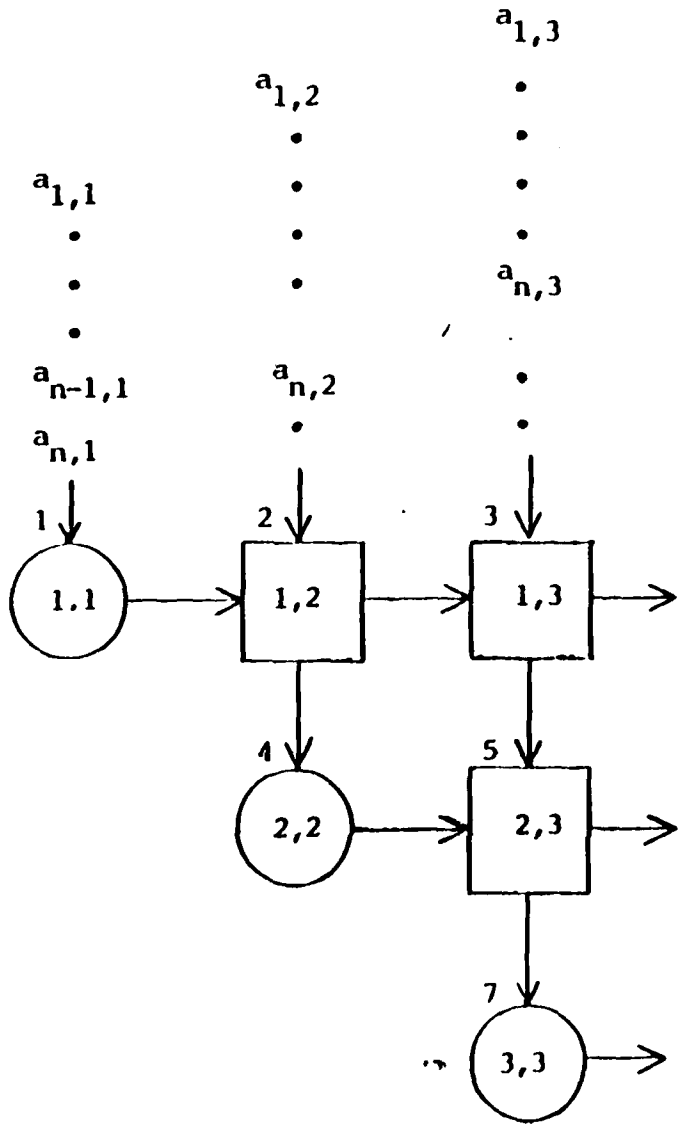
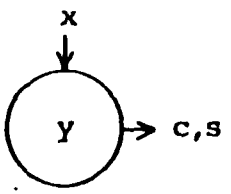
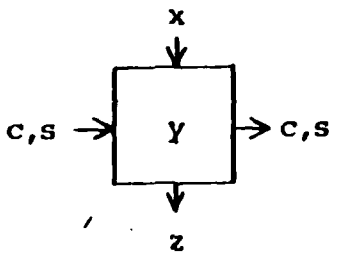


Figure 1. Gentleman/Kung array for QU





$$c = x \,/\, \sqrt{x^2 + y^2}$$

$$s = y \,/\, \sqrt{x^2 + y^2}$$

$$y = \sqrt{x^2 + y^2}$$

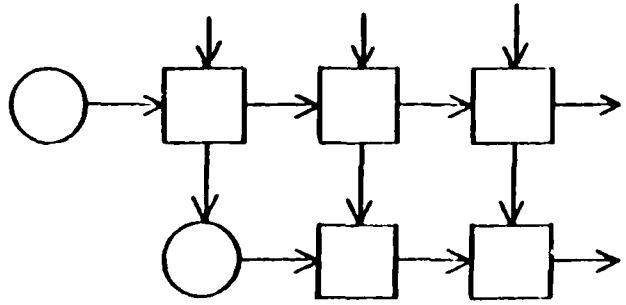
$$y = cx + sy$$

$$z = -sx + cy$$

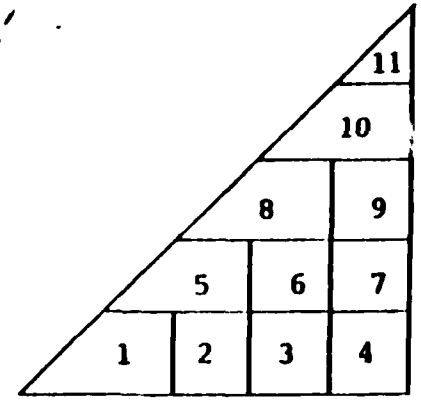
Figure 2. Cells of the QU array

p = 2

q = 4



Trapezoidal subarray



Schedule of passes for
QU factorization

Figure 3. Simulation of the full array by a subarray

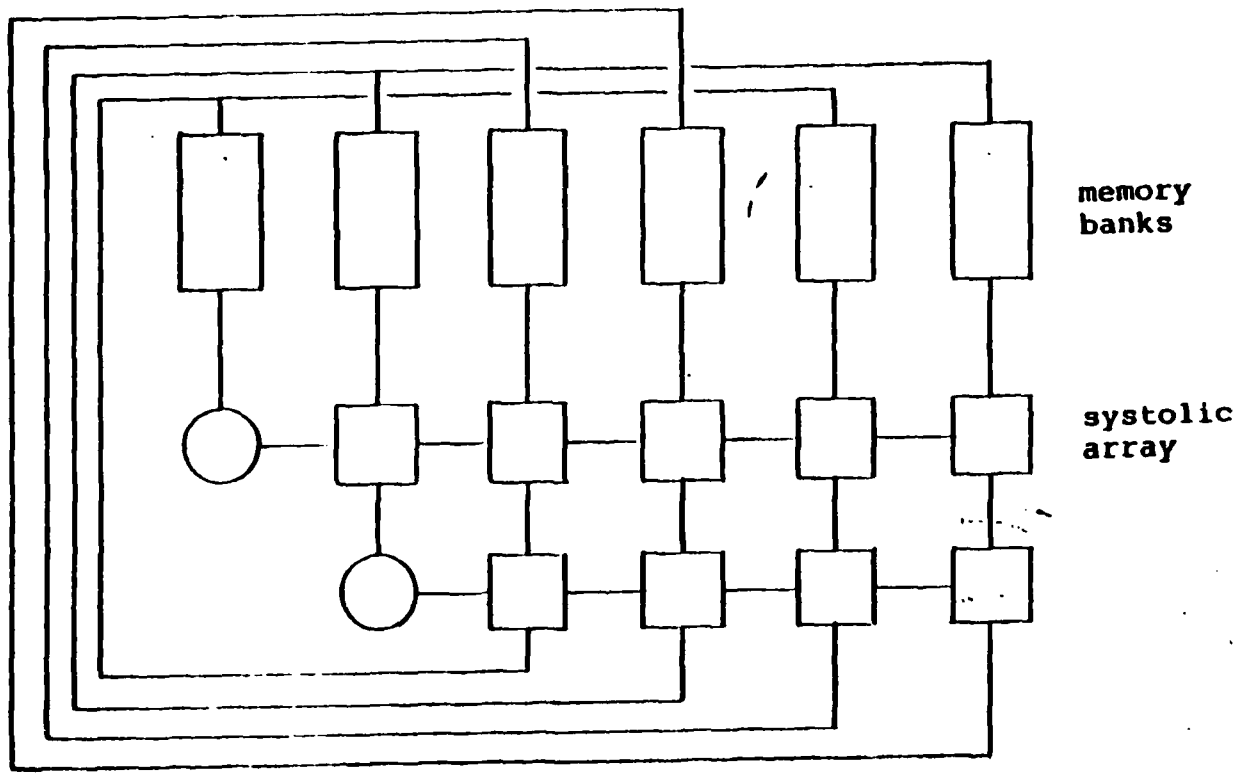


Figure 5. QU memory support system

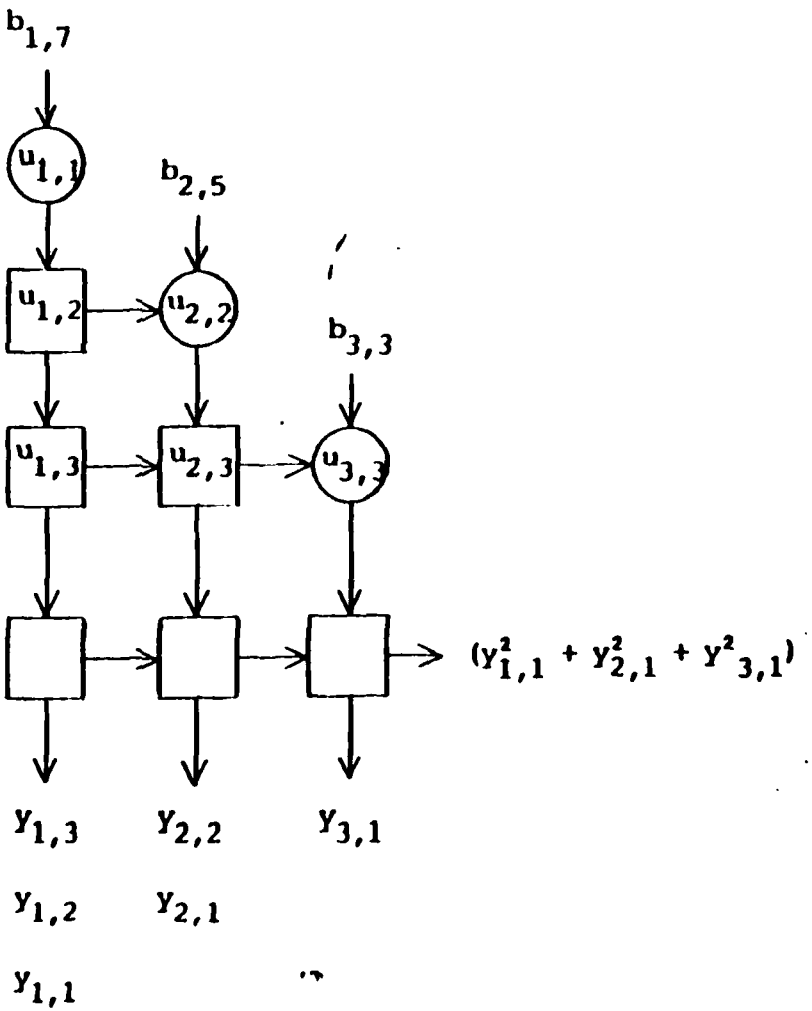


Figure 6. The backsolve array

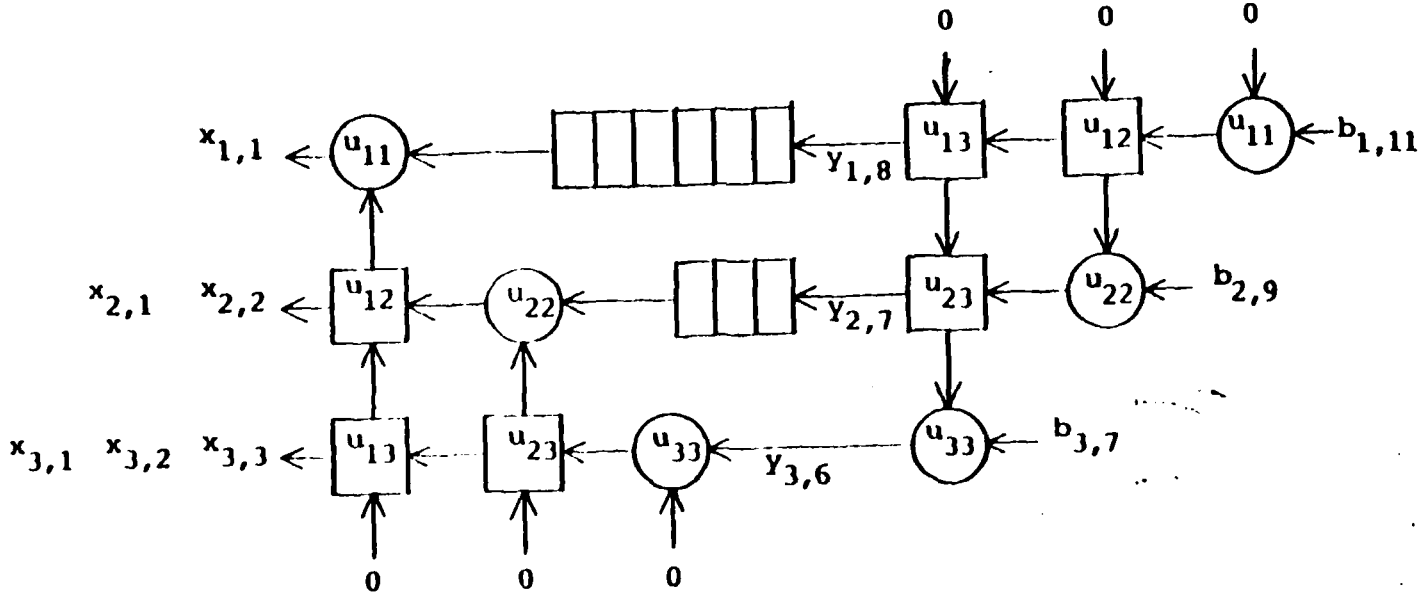


Figure 7. Two-array forward and backsolve

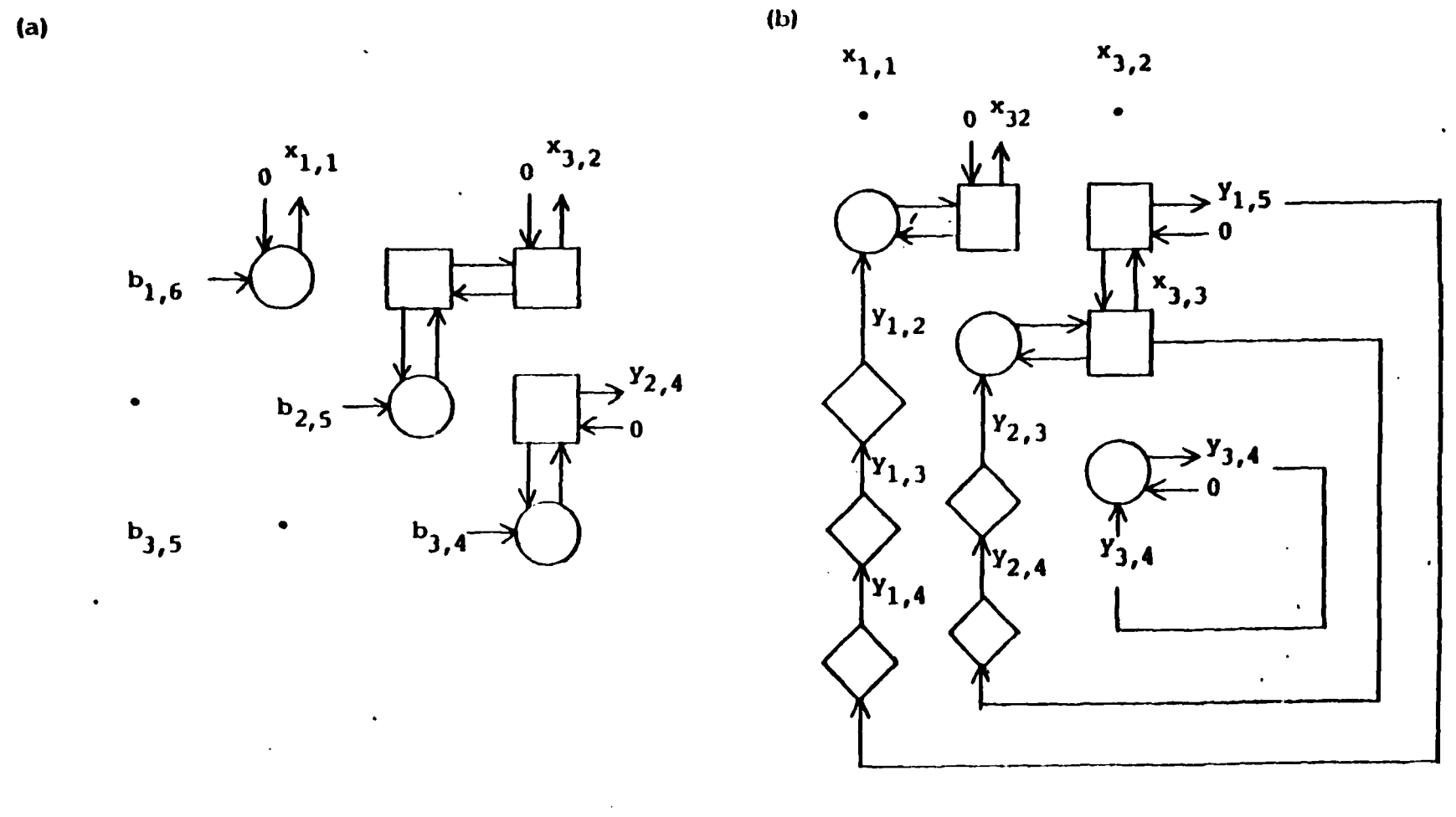
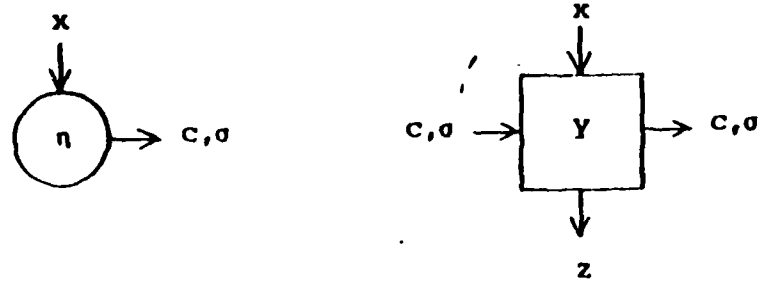


Figure 8. Single array for forward and backsolve; (a) odd cycle, (b) even cycle



$\rho = \sqrt{\eta^2 + |x|^2}$

$c = x/\rho$

$\sigma = \eta/\rho$

$\eta = \rho$

$y = c^* x + \sigma y$

$z = -\sigma x + c y$

Figure 9. Cells for complex QU factorization

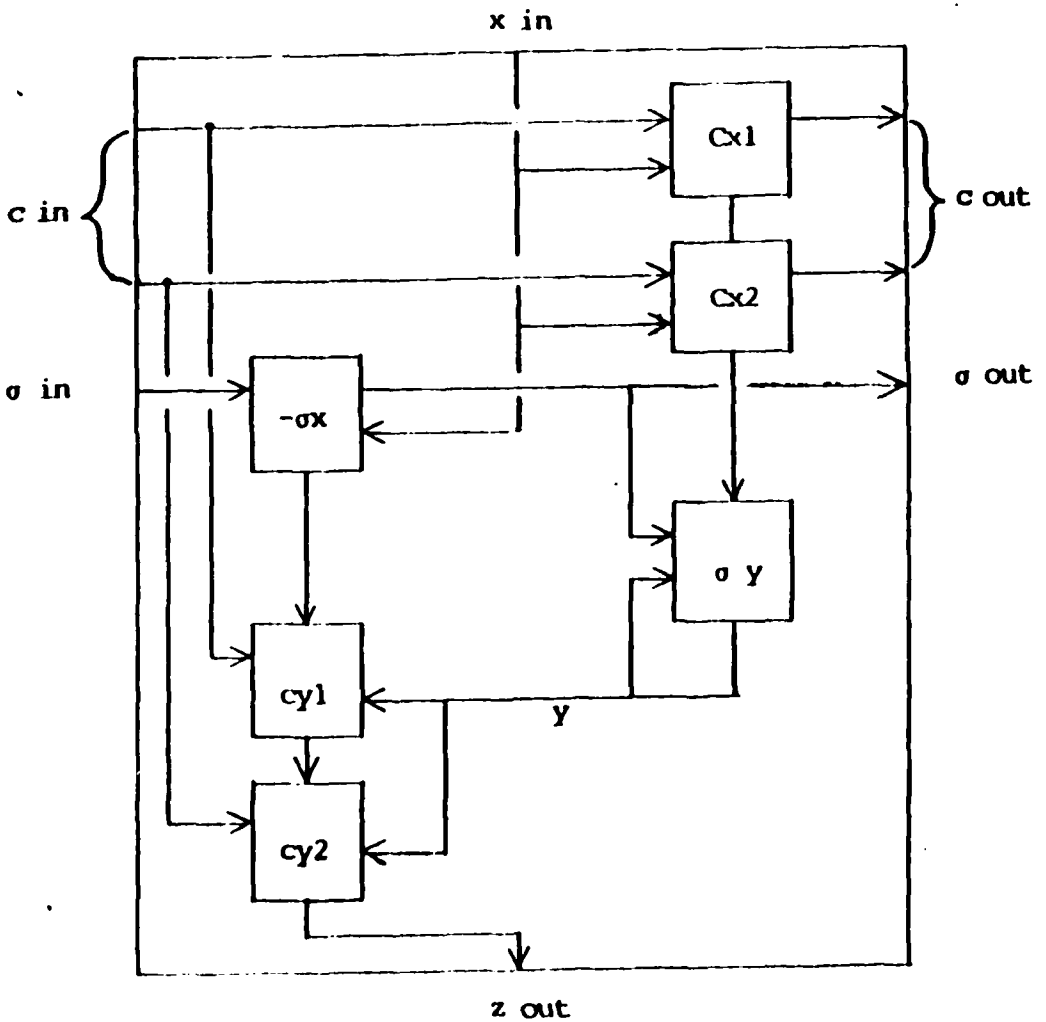


Figure 10. Complex Givens cell

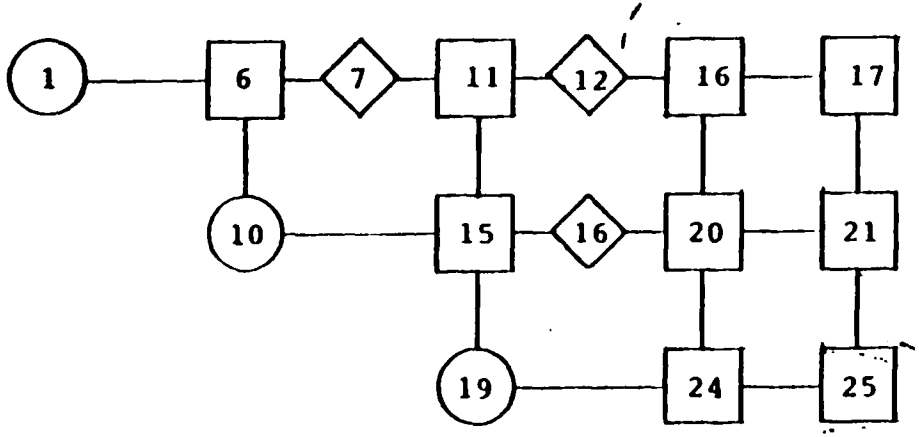


Figure 11. Effect of boundary cell latency

TRITA-NA-8313

Inst för Numerisk analys och datalogi
KTH
100 44 Stockholm

Dept of Numerical Analysis and
Computing Science
The Royal Institute of Technology
S-100 44 Stockholm, Sweden

ON SYSTOLIC ARRAYS FOR UPDATING THE CHOLESKY FACTORIZATION

by

Robert Schreiber *
Wei-Pai Tang **

TRITA-NA-8313

* Department of Numerical Analysis and Computing Science,
The Royal Institute of Technology, Stockholm, Sweden
Permanent address is:

** Department of Computer Science, Stanford University, Stanford, CA 94305, USA

1. Summary

Every symmetric positive definite n×n matrix A admits both the Cholesky factorization

$$A = R^T R \tag{1.1}$$

where R is upper triangular and the related "square-root-free" factorization

$$A = LDL^T \tag{1.2}$$

where L is unit-lower triangular and D is diagonal with positive elements.

In a number of situations, it is necessary to factor the modified matrix

$$\bar{A} = A + \alpha zz^T. \tag{1.3}$$

Gill, Golub, Murray, and Saunders (hereafter GGMS) give several algorithms for modifying the factors of A [2]. The algorithms all require $Cn^2 + \mathcal{O}(n)$ operations for some constant C; direct Cholesky factorization requires $(1/6)n^3 + \mathcal{O}(n^2)$ operations, where an operation is one multiplication and one addition. The purpose of this paper is to describe the implementation of some of these algorithms by systolic array.

The principle application for systolic computation of updated Cholesky factors comes from digital signal processing. There, A is an estimate of the covariance matrix of a random signal vector. Periodically it is updated according to (1.3); in these applications α is ordinarily positive.

In this context one may also have to consider rank k modifications

$$\bar{A} = A + \alpha_1 z_1 z_1^T + \ldots + \alpha_k z_k z_k^T. \tag{1.4}$$

We discuss an appropriate systolic method for this situation in Section 3.

1.1 Notation. The elements of a matrix A and a vector x shall be denoted by a_{ij} and x_i. The symbols R, L and D shall be used for upper triangular, unit lower triangular and diagonal matrices, respectively. $D = \mathrm{diag}(d_1, \ldots, d_n)$. We write e_i for the i^{th} column of the identity matrix.

We shall use the notation P_i^j for a plane rotation matrix that differs from the identity matrix only in that

$$p_{ii} = p_{jj} = c$$

$$p_{ij} = -p_{ji} = s$$

where $c = \cos\theta$, $s = \sin\theta$. Given i, j and x there exists θ such that

$$\left(P_i{}^j x\right)_k = \begin{cases} x_k & , k \neq i,j \\ 0 & , k = i \\ (x_i^2 + x_j^2)^{\frac{1}{2}} & , k = j \end{cases}$$

2. Methods using the Cholesky factorization of $D + \alpha pp^T$

Let us determine the factors

$$\bar{A} = \bar{L}\bar{D}\bar{L}^T$$

By (1.2),

$$\bar{A} = A + \alpha zz^T = L(D + \alpha pp^T)L^T$$

where $Lp = z$. First, find p by back-substitution. If we find the factorization

$$D + \alpha pp^T = \tilde{L}\tilde{D}\tilde{L}^T . \tag{2.1}$$

Then the modified factorization of

$$\bar{A} = L\tilde{L}\tilde{D}\tilde{L}^T L^T$$

is given by

$$\bar{L} = L\tilde{L}, \quad \bar{D} = \tilde{D}. \tag{2.2}$$

We consider two algorithms of GGMS for computing (2.1) - (2.2). The first is readily implemented by systolic array. It may fail, however, when $\alpha < 0$ and $\bar{A}$ is ill-conditioned. The second is infallible, but not easily implemented.

GGMS show that, in (2.1),

$$\tilde{\ell}_{rs} = p_r \beta_s, \quad 1 \leq s < r \leq n . \tag{2.3}$$

Thus, to get the factorization (2.1), we need only find β_j, $\tilde{d}_j$, $1 \leq j \leq n$. The special structure of $\tilde{L}$ also allows fast computation of $\bar{L} = L\tilde{L}$. Define the vectors

$$w_r^{(j)} = \sum_{i=j}^{r} \ell_{ri} p_i = z_r - \sum_{i=1}^{j-1} \ell_{ri} p_i, \quad r = j, j+1, \ldots, n.$$

These values are generated in the course of back-substitution to find p. When computing $\bar{L}$ they are used again. By (2.2), (2.3)

$$\begin{aligned} \bar{\ell}_{rj} &= \sum_{i=j}^{r} \ell_{ri} \tilde{\ell}_{ij} \\ &= \ell_{rj} + \sum_{i=j+1}^{r} \ell_{ri} \tilde{\ell}_{ij} \\ &= \ell_{rj} + \sum_{i=j+1}^{r} \ell_{ri} p_i \beta_j \\ &= \ell_{rj} + w_r^{(j+1)} \beta_j . \end{aligned}$$

The final recurrence for computing $\tilde{L}$ and $\tilde{D}$ is this:

Algorithm 2.1

1. Let $\alpha_1 = \alpha$; $w^{(1)} = z$.

2. for $j = 1,2, \ldots ,n$, compute

$$p_j = w_j^{(j)} \tag{2.4}$$

$$\bar{d}_j = d_j + \alpha_j p_j^2 \tag{2.5}$$

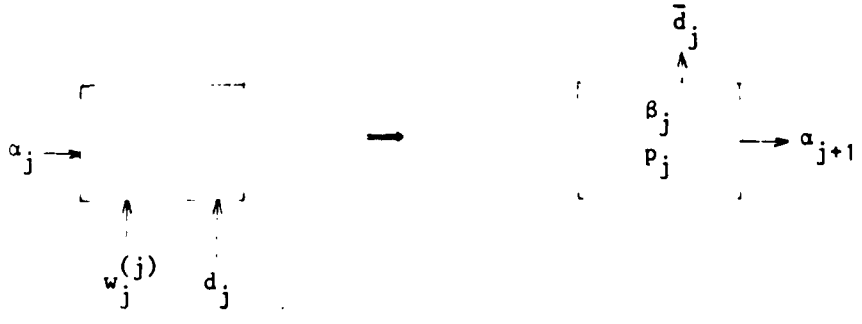
$$\beta_j = p_j \alpha_j / \bar{d}_j \tag{2.6}$$

$$\alpha_{j+1} = d_j \alpha_j / \bar{d}_j \tag{2.7}$$

$$\left.\begin{aligned} w_r^{(j+1)} &= w_r^{(j)} - p_j \ell_{rj} \qquad (2.8)\\ \bar{\ell}_{rj} &= \ell_{rj} + \beta_j w_r^{(j+1)} \qquad (2.9)\end{aligned}\right\} \quad r = j+1, \ldots ,n.$$

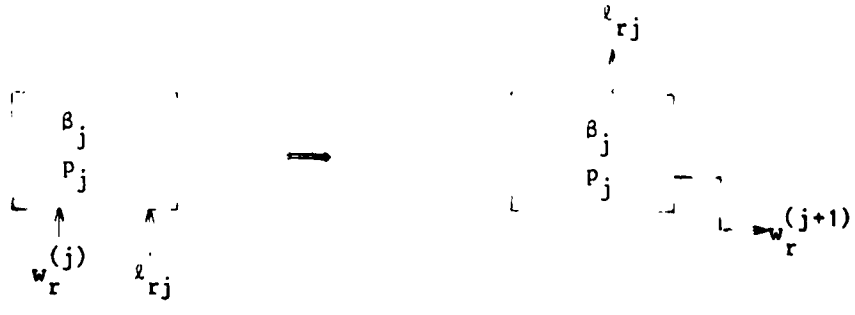
We now consider systolic array methods. First, there is an obvious method, shown in Figure 1, in which there is a processor for each stage $1 \le j \le n$ in the algorithm. The cell used is this

first clock ($r = j$):



(see (2-4) - (2.7));

subsequent clocks ($r > j$):



(see (2.8), (2.9)).

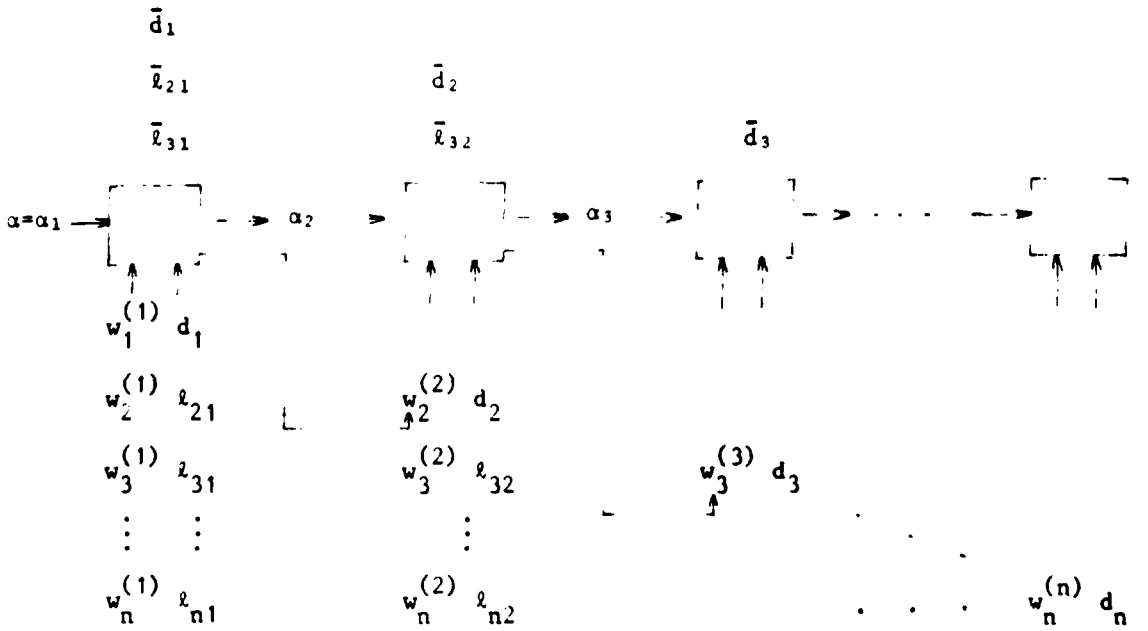
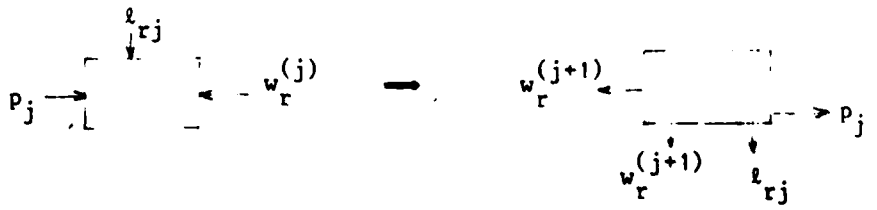


Figure 1. An array for Cholesky update

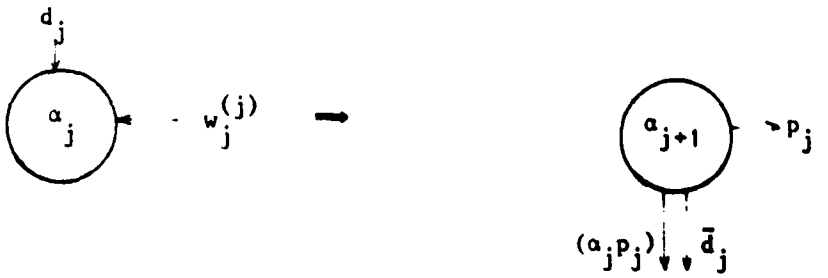
The principle disadvantage of this design is that the cells are relatively complicated. All must divide. All have memory.

There is a second array, shown in Figure 2, without these disadvangages.

The cells are defined as follows

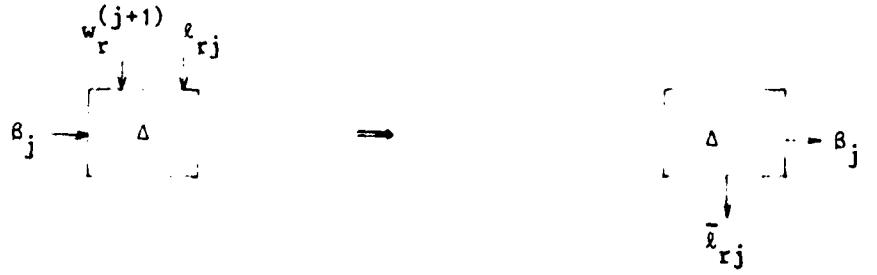


(see (2.8));



(see (2.4), (2.5), (2.7));

(see (2.6));



(see (2.9)).

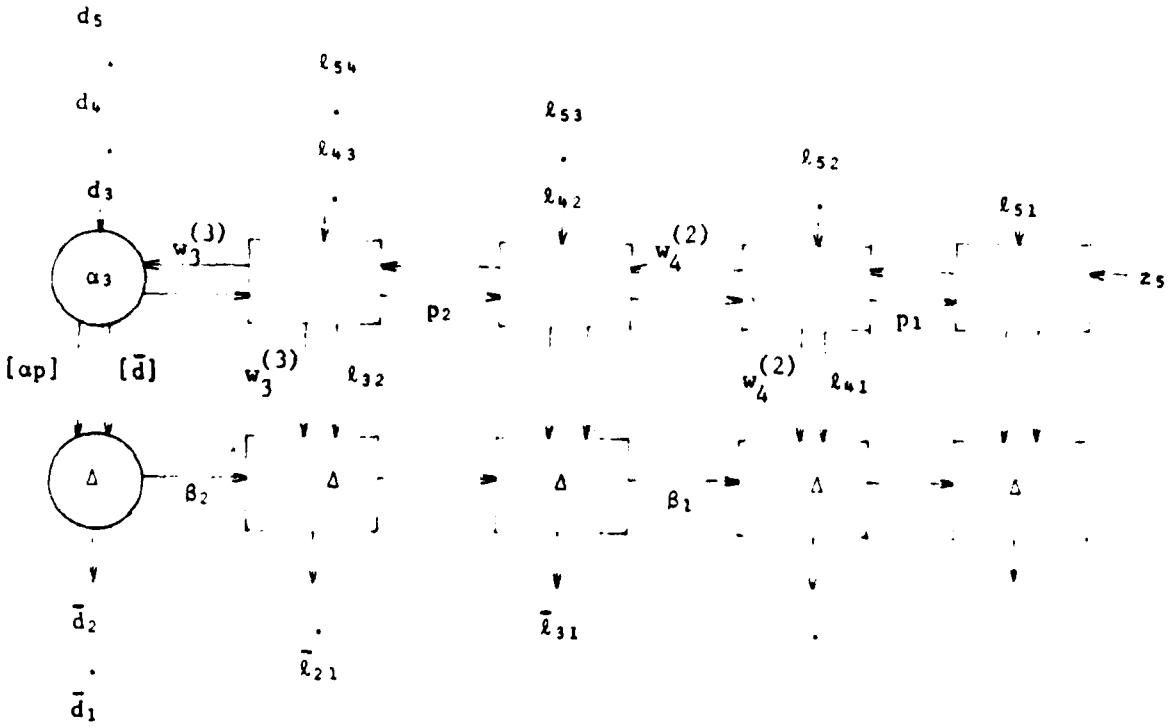


Figure 2. The second array for algorithm 1

Algorithm 2.1 may fail. GGMS point out that when $\alpha < 0$ and $\bar{A}$ is nearly singular, rounding error can cause one of the computed values $\bar{d}_j$ to be nonpositive, so that an indefinite factorization is obtained.

GGMS have proposed a modified algorithm, algorithm 2.2 below, that is guaranteed to provide a positive definite factorization.

Algorithm 2.2

1. Solve $LP = z$

2. Define

$$\alpha_1 = \alpha$$

$$s_1 = \sum_{j=1}^{n} p_j^2/d_j$$

$$\sigma_1 = \alpha/[1 + (1 + \alpha s_1)^{\frac{1}{2}}].$$

3. for $j = 1,2, \ldots ,n$, compute

(a) $q_j = p_j^2/d_j$

(b) $\theta_j = 1 + \sigma_j q_j$

(c) $s_{j+1} = s_j - q_j$

(d) $\rho_j^2 = \theta_j^2 + \sigma_j^2 q_j s_{j+1}$

(e) $\bar{d}_j = \rho_j^2 d_j$

(f) $\beta_j = \alpha_j p_j/\bar{d}_j$

(g) $\alpha_{j+1} = \alpha_j/\rho_j^2$

(h) $\sigma_{j+1} = \sigma_j(1 + \rho_j)/[\rho_j(\theta_j + \rho_j)]$

(i) for $r = j+1, j+2, \ldots ,n$,

$$w_r^{(j+1)} = w_r^{(j)} - p_j \ell_{rj}$$

$$\bar{\ell}_{rj} = \ell_{rj} + \beta_j w_r^{(j+1)} \qquad (2.16)$$

This algorithm does not permit realization using one pass of L through an array, as does Algorithm 2.1. The reason is that s_1 is needed before step 3 can begin, and the full backsolve in step 1 must be completed in order to compute s_1. There is a relatively obvious two-pass method, in which p, w, q, and s_1 are computed from L and z in the first pass through the array, and all other quantities in the second pass. The boundary cell computation for the second pass is quite complex, as it encompasses steps 3(b) - 3(h). In practice, it will be rather difficult to match the speed of such a boundary cell to that of the rest of the array, which would consist of a linear array, each cell

performing the computation (2.10) for a fixed value of r - j, exactly as in the lower half of the array in Figure 2.

3. Methods based on plane rotations

GGMS give several algorithms using plane rotations for modifying Cholesky factorizations. We have found two of the more efficient of these to be easily implementable as systolic arrays. The first is simple, efficient, requires only one pass of the data through the array, but works only when $\alpha > 0$. The second works in general, but requires two passes. Both can be extended to handle rank - k updates (1.4).

3.1 Positive rank - 1 changes

We compute the modified factorization

$$\bar{R}^T\bar{R} = R^TR + \alpha zz^T, \quad \alpha > 0,$$

by reducing the matrix $[\alpha^{\frac{1}{2}}z \vdots R^T]$ to lower triangular form,

$$[\alpha^{\frac{1}{2}}z \vdots R^T]P = [\bar{R}^T \vdots 0]$$

where $P = P_2^1\, P_3^2 \ldots P_{n+1}^n$. It follows that

$$\bar{R}^T\bar{R} = [\alpha^{\frac{1}{2}}z \vdots R^T]\, P P^T \begin{bmatrix} \alpha^{\frac{1}{2}}z^T \\ \hline R \end{bmatrix} = R^TR + \alpha z z^T,$$

so that $\bar{R}$ is the updated Cholesky factor.

The method can be readily extended to handle rank - k updating efficiently. In this case

$$\bar{A} = A + ZAZ \tag{3.1}$$

where

$$Z = [z_1, z_2, \ldots, z_k]$$

is an n×k matrix and

$$A = \mathrm{diag}(\alpha_1, \alpha_2, \ldots, \alpha_k).$$

When $\alpha_j > 0$, $1 \le j \le k$ then we can form the matrix

$$\begin{bmatrix} A^{\frac{1}{2}}Z^T \\ \hline R \end{bmatrix}$$

and reduce it to upper triangular form by premultiplication by a suitable sequence

$$P = P^{(n)}\ P^{(n-1)}\ \ldots\ P^{(1)}$$

$$P^{(j)} = P_{j+1}^{\ j}\ P_{j+2}^{\ j+1}\ \ldots\ P_{k+j}^{\ k+j-1}$$

of plane rotations.

The advantage of considering k updates together instead of as a sequence of k rank-1 updates is that the modification of R can be carried out by one pass of the data through a systolic array of $O(kn)$ processors. Thus, while the number of arithmetic operations is the same, the total time and the number of I/O operations (including memory references) is less by the factor k.

We postpone consideration of the array until the next section.

3.2 Negative rank-1 changes

GGMS have given a modification of the previous method for the case $\alpha < 0$. We shall now generalize their approach to the case of a rank-k change. Let

$$\bar{A} = A - ZAZ^T = R^TR - ZAZ^T \tag{3.2}$$

where Z is n×k and $A > 0$ is diagonal. For convenience we take $A = I$, without loss of generality. Now let P be the n×k solution to

$$R^TP = Z. \tag{3.3}$$

It is easy to show that $\bar{A}$ remains positive definite if and only if $I - P^TP$ is positive definite. Now form the matrix

$$\begin{bmatrix} P & R \\ \cdots & \cdots \\ D_n & 0 \end{bmatrix}$$

(D_n is a lower triangular matrix to be specified below) and premultiply by an orthogonal matrix Q of the form $Q = P^{(1)}\ P^{(2)}\ \ldots\ P^{(k)}$ where $P^{(j)} = P_{n+j}^{\ 1}\ \ldots\ P_{n+j}^{\ n}$ zeros column j of P and leaves R in upper-triangular form, until

$$Q\begin{bmatrix} P & R \\ \cdots & \cdots \\ D_n & 0 \end{bmatrix} = \begin{bmatrix} 0 & \bar{R} \\ \cdots & \cdots \\ D_C & S^T \end{bmatrix}. \tag{3.4}$$

From (3.4) we conclude that

$$P^TP + D_n^TD_n = D_0^TD_0 \tag{3.5}$$

$$R^TR = \bar{R}^T\bar{R} + SS^T \tag{3.6}$$

$$R^TP = SD_0. \tag{3.7}$$

From (3.3) and (3.7) we have that

$$Z = SD_0. \tag{3.7'}$$

We now choose D_n. Take D_n to be the Cholesky Factor

$$D_n^TD_n = I - P^TP. \tag{3.8}$$

Thus, by (3.5), we have that $D_0 = I$. Hence, by (3.7') and (3.6)

$$\bar{R}^T\bar{R} = R^TR - ZZ^T.$$

3.3 Arrays for updating with plane rotations

We first consider the method of section 3.1 for positive updates. The array presented is a generalization of an array of Heller and Ipsen for QR factorization of a banded matrix [3]. A similar notion also appears in Schreiber's work on systolic arrays for the eigenvalues of a nonsymmetric matrix [5].

The array is shown in Figure 3. It is $k \times (n+k)$, rectangularly connected. Not all of the $k(n+k)$ cells actually do anything, as is shown in the figure. If the first elements of R and Z enter at time 1, then the first element of $\bar{R}$, $\bar{r}_{11}$, emerges at time $2k+1$, and the last, $\bar{r}_{nn}$ at time $2(n+k) - 1$.

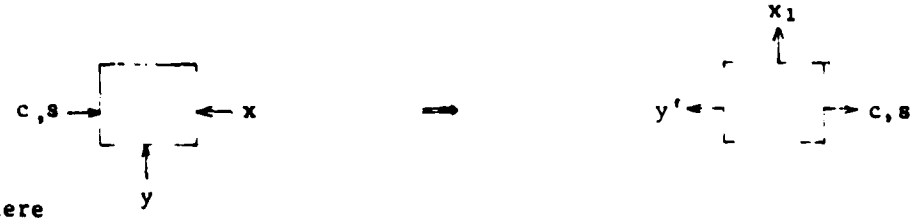
Again, for convenience, we assume $\alpha = 1$. The cells of this array operate as follows.

boundary cell:

where $c = \cos\theta$, $s = \sin\theta$, and

$$\begin{pmatrix} x' \\ 0 \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix};$$

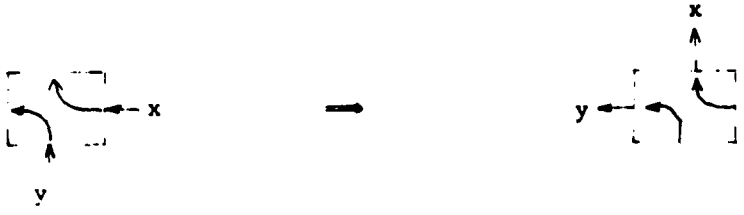
internal cell:



where

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} ;$$

delay cell:



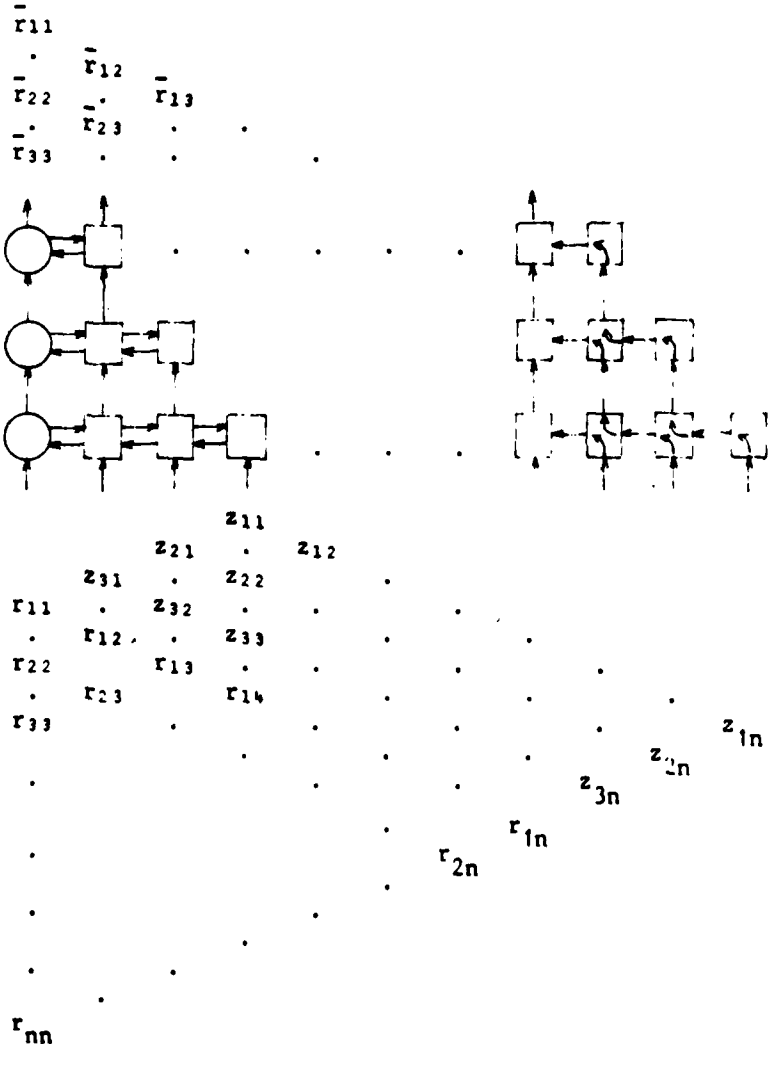


Figure 3. The Heller-Ipsen array for $\alpha > 0$, $k = 3$

It would be quite possible to eliminate the delay cells and provide input data directly at the right-hand edge of the remaining $k \times n$ array. The pattern of access to the data is slightly more complex when this is done.

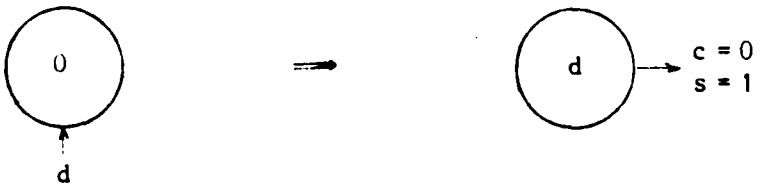
The second algorithm, for $\alpha < 0$, requires two passes over the matrix R. First, the systems (3.3) are solved using a $k \times n$ array, an obvious generalization of Kung and Leiserson's array for $k = 1$ and banded R [4]. Next, we compute D_n off-line. We assume that k is quite small, so that it is easy to do this. The matrix $I - P^T P$, of which D_n is the Cholesky factor, can be accumulated by another, rather obvious, array of $k(k+1)/2$ cells.

Now we consider the reduction (3.4). This is done by the array shown in Figure 4.

The matrices D_n, P, and R enter in the format shown, with $D = [d_{ij}]$. The last element, $r_{1,1}$, enters $2n + k - 1$ clocks after the first, and leaves from the top k clocks later. The matrices $D_0 = I$ and S reside in the array. The cells used are these:

<u>boundary cell</u>:

if empty:



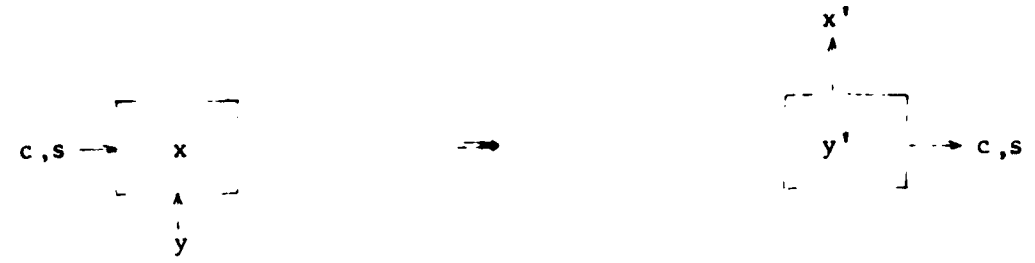
otherwise

where $c = \cos\theta$, $s = \sin\theta$, and

$$\begin{pmatrix} d' \\ o \end{pmatrix} = \begin{pmatrix} c & s \\ s & -c \end{pmatrix} \begin{pmatrix} d \\ p \end{pmatrix};$$

[there is really nothing special about the "empty" case]

internal cell:



where

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} c & s \\ s & -c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

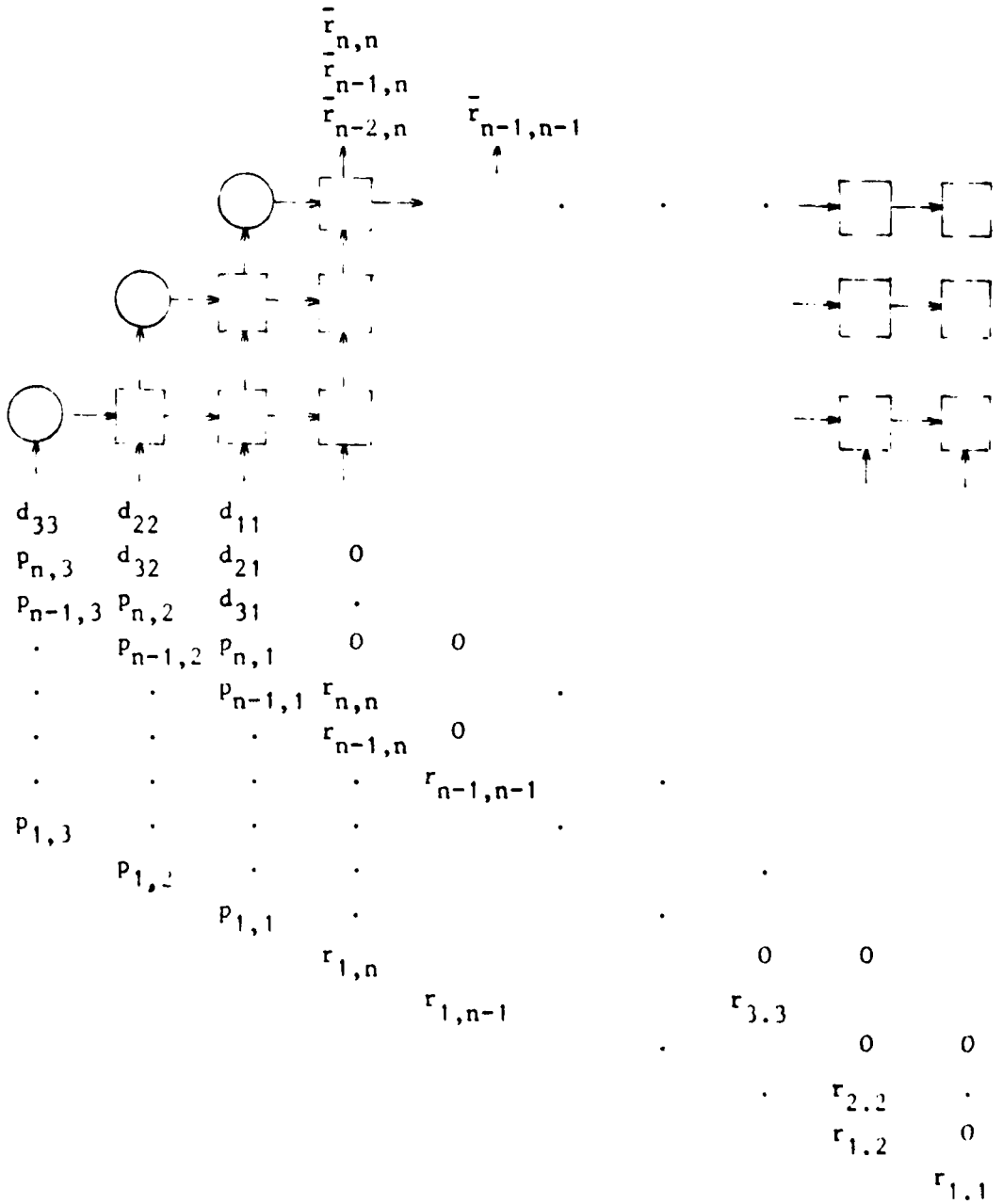


Figure 4. Updating with plane rotations when $\alpha < 0$

Initially all cells contain zero and all rotations are $c = 0$, $s = 1$.

In many respects this array is like a $k \times n$ section of the Gentleman-Kung array for QR factorization [1], which was described more fully by Schreiber and Kuekes [6].

4. Related problems

The problem of updating Cholesky factors frequently arises in contexts in which

$$A = B^T B$$

for a rectangular matrix B of n columns. In the orthogonal factorization

$$B = QR$$

the matrix R is the Cholesky factor of A. If a row z^T is appended to B this causes a rank-one change to A. To update R, any of the methods discussed here can be used. The first method of Section 3 is particularly appropriate since it uses orthogonal operations to update R. If Q is to be updated, too, one computes

$$Q_I \begin{bmatrix} Q & 0 \\ \cdots & \cdots \\ 0 & 1 \end{bmatrix} \begin{bmatrix} B \\ \cdots \\ z^T \end{bmatrix} = Q_I \begin{bmatrix} R \\ \cdots \\ z^T \end{bmatrix} = \begin{bmatrix} \bar{R} \\ \cdots \\ 0 \end{bmatrix}$$

so that

$$\bar{Q} \leftarrow Q_I \begin{bmatrix} Q & 0 \\ \cdots & \cdots \\ 0 & 1 \end{bmatrix}.$$

The array used can be enlarged to allow this computation to be performed also.

Acknowledgements. The work of the first author was supported in part by ESL, Inc., in part by the US Office of Naval Research under Contract Nr. N00014-82-K-0703, and in part while a visiting researcher at the Royal Institute of Technology in Stockholm and at Uppsala University, Sweden.

We thank Philip Kuekes of ESL and Norman Owsley of the Navy Underwater Systems Center for provoking our interest in this problem.

References

[1] W.M. Gentleman and H.T. Kung. Matrix triangularization by systolic array.
Proc. SPIE, Vol. 298: Real-Time Signal Processing IV, Society of Photo-optical Instrumentation Engineers, Bellingham, Washington, 1981.

[2] P.E. Gill, G.H. Golub, W. Murray, and M.A. Saunders. Methods for modifying matrix factorizations.
Math. Comp. 28, pp. 505-535 (1974).

[3] D.E. Heller and I.C.F. Ipsen. Systolic networks for orthogonal equivalence transformations and their applications.
Proc. 1982 Conf. on Advanced Research in VLSI, MIT, Cambridge, Massachusetts, 113-122, 1982.

[4] H.T. Kung and C.E. Leiserson. Systolic arrays for VLSI. In C.A. Mead and L.A. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980.

[5] R. Schreiber. Systolic arrays for eigenvalues.
Proc. Inter-American Workshop on Numerical Methods (Caracas 1982). Springer-Verlag, to appear.

[6] R. Schreiber and P. Kuekes. Systolic linear algebra machines in digital signal processing.
Proc. USC Workshop on VLSI and Modern Signal Processing (Los Angeles, 1982). Prentice-Hall, to appear.

SYSTOLIC ARRAYS: HIGH PERFORMANCE PARALLEL MACHINES FOR MATRIX COMPUTATION

Robert Schreiber[1]

I. INTRODUCTION

In this paper we shall summarize the recent development of systolic array methods for some of the important standard problems in numerical linear algebra. We shall discuss LU and QR factorizations, eigenvalue problems, and the singular value decomposition. All the work we shall describe has been done since 1981. Our aim is to introduce the reader to this rapidly developing branch of numerical computation.

Systolic arrays were introduced and named by Kung and Leiserson [12]. Although their ideas had antecedents (see [8] for example), they first fully realized and proclaimed that these designs--highly parallel computing networks with regular data flows, two-dimensional lattice form, simple identical cells, regular input and output patterns, and heavy reuse of data--were an ideal way to use the emerging VLSI technology to obtain very high performance for suitable computations.

After these first promising designs, which implemented a few relatively simple matrix computations, two important questions were open. Could systolic arrays be found for a broader class of matrix problems, including the more important and difficult matrix decompositions? And could they be integrated into real computing systems without introducing incapacitating losses of efficiency?

---

[1]Department of Computer Science, Stanford University, Stanford, California 94305

The first of these questions has now been answered and the answer is a definite "yes." This research has done more than answer the question of the applicability of the systolic array idea. It is, rather, the beginning of an intriguing new approach to numerical computation. In this new approach the flow of data in both time and space is important, whereas traditionally only the sequence of computations in time was considered. An interesting facet is that the standard algorithms have not always proven suitable for systolic array realization. For instance, for symmetric eigenvalue problems, Brent and Luk [5] have found that Jacobi's method is better than QR. Their method uses a permutation of the off-diagonal elements that seems to be better than the usual cyclic-by-rows permutation [5].

The second question is being answered now. Several projects for building VLSI based systolic hardware are currently in progress or complete ([1], [19]). An effort at ESL, Inc., should produce a systolic machine with performance in the 100-1000 megaflop range in the next few years.

It appears at the moment that digital signal and image processing [15], rather than standard scientific computing and elliptic equation solving in particular, offers the sort of problem best suited for systolic array solution. Nevertheless these ideas can be usefully applied to elliptic equations. In Section VI we shall discuss an implementation of multigrid methods by highly parallel computing networks.

II. LU AND QR FACTORIZATIONS

The first such array was an $m \times m$ hexagonally connected array for LU factorization of $n \times n$ banded matrices with bandwidth m in time $O(3n)$ [12]. Unfortunately, there was no provision for pivoting to enhance stability. Next, an array for QR or LU factorization of a dense square matrix, in linear time, was given by Bojanczyk, Brent, and Kung [2]. A different array, better in that it could handle rectangular $m \times n$ matrices in time $O(m)$, was given by Gentleman and Kung [10]. The unit of time we use is the cycle time of a cell of the given array. In this time, every array accepts one set of

inputs and produces one set of outputs. Finally, Heller and Ipsen developed a rectangular array for band matrix LU and QR factorization [11]. (See Figures 1 and 2.) Many practical details concerning the implementation of the Gentleman-Kung array were discussed by Schreiber and Kuekes; they also give an array for triangular systems with many right-hand sides [25].

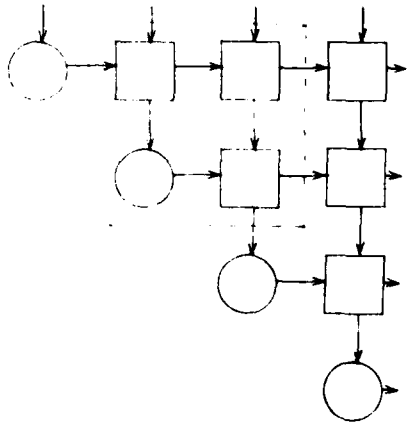


Figure 1.
The Gentleman-Kung Array

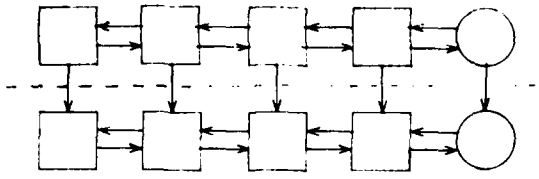


Figure 2.
The Heller-Ipsen Array

The QR factorization arrays employ Givens rather than Householder transformations. The LU arrays employ a stable variant of Gaussian elimination that uses "neighbor pivoting." When the matrix element a_{ij} is eliminated, the elimination is done by subtracting from row i a multiple of row i-1. These rows are first exchanged, if necessary, to make the multiplying factor smaller in modulus than one. Partial pivoting does not lend iteself to systolic implementation.

III. EIGENVALUE COMPUTATION

The obvious first approach was to attempt to implement the standard algorithm for the symmetric problem: reduction to tridiagonal form by orthogonal similarity transformation

followed by a method, QR iteration for example, for the tridiagonal matrix. A linear array that requires $O(n^2)$ time is easy to find, but more parallelism has so far proved elusive. Schreiber showed, however, that reduction to a matrix with 2k+1 non-zero diagonals could be done in n^2/k cycle times using a $k \times n$ sub-array of the Gentleman-Kung array. The eigenvalues of the banded matrix can be found using QR iteration which was shown to be implementable by the Heller-Ipsen array in only a little more time than is needed for QR factorization [13]. If further speed is desired, then up to n/k QR iterations can be performed simultaneously by a pipeline of Heller-Ipsen arrays. Choosing $k = n^{1/2}$ yields a method using $O(n^{3/2})$ processors and $O(n^{3/2})$ time. (There is some difficulty concerning the choice of shifts in the QR iteration when iterations are pipelined.) These techniques are also applicable to nonsymmetric matrices [14].

The pursuit of a linear-time solution led Brent and Luk to consider Jacobi methods. They have found an implementation using an $n/2 \times n/2$ array, which implements a cyclic Jacobi sweep in n-1 cycle times [5]. Their experiments have shown that O(log n) sweeps are required for convergence. In practice, for $n \leq 1000$, no more than 10 sweeps would be required [5].

IV. SINGULAR VALUE DECOMPOSITION (SVD)

To both the approaches of Schreiber and Brent-Luk there are analogous approaches to the SVD. Schreiber's takes mn/k cycle times to reduce an $m \times n$ matrix to an upper triangular matrix with k+1 nonzero diagonals. QR iteration can then be carried out using Heller-Ipsen arrays [16]. Brent and Luk employ a linear array, and their method requires O(mn log n) time [4]. Brent, Luk, and Van Loan have just reported an improved method requiring almost linear time [6].

One of the uses of SVD is in solving ill-conditioned least-squares problems. An alternative method is to construct the generalized inverse of a rank-deficient matrix closest to the given matrix by an iterative method. An algorithm of Ben-Israel, improved by Schreiber, can be used. The generalized

inverse is obtained in $O(m \log \text{cond}(A))$ time using n^2 processors [18].

V. DECOMPOSING ARRAYS

A most important question from the practical point of view is this: can a physical array of fixed size (number of cells) be used to solve problems of arbitrary size? The question is important because, obviously, problems of widely differing sizes may be confronted in some applications. Not so obviously, in some applications the problems may have only one size, but they may occur infrequently enough that a full-sized array would not be kept busy for more than a fraction of the time. In these cases it would be more efficient to build a smaller array and have it simulate a full-sized array of the same kind. The question is whether this hardware/time tradeoff is possible.

For some arrays it is easy to see how to do this. Consider the Gentleman-Kung array (Figure 1). Suppose a subarray of the size shown within the dashed outlines is available. The sub-array could be used to perform the work done by the outlined part of the whole array because all the data flowing into that part of the array is known at the outset. Data flowing out of the subarray has to be stored. The sub-array can next be moved to cover another part of the large array for which all inputs are now known. This continues until the whole array has been covered.

For some arrays, the only sub-array for which all inputs are known is the entire array. The Kung-Leisserson LU factorization array is an example. Such an array is not "decomposable" by this technique. We consider that this is a serious drawback.

The Heller-Ipsen array (Figure 2) is partially decomposable. If we cut it by a horizontal line (the dashed line in Figure 2) then, since all data flows through the cut from bottom to top, we can "run" the bottom part of the array, save the output, then run the top part using this saved data as input. But if a vertical cut is made, data flows across the cut in both directions. Moreover, these data cannot be known

unless the computations performed by both the left and right halves of the array are done.

The Brent-Luk SVD array at first appears to be indecomposable. Despite this we have recently found a way to use this array to solve larger problems. The key is to use a p-processor Brent-Luk array as the basic "cell" in a q-processor "super-array" that works on matrices with 2pq columns. Columns move between these cells in groups of p. The idea also applies to the Brent-Luk eigenvalue array. We shall give the details in a later paper [19].

VI. MULTIGRID METHODS

To illustrate the applicability of systolic-array-like devices for elliptic problems, we consider implementation of multigrid algorithms. Full details are given by Chan and Schreiber [7]. Related work is reported by Brandt [3] and Gannon and Van Rosendale [9]. Consider a standard multigrid algorithm with n^d gridpoints on the finest grid and a coarse-to-fine mesh-length ratio of a, in which c coarse grid interations are done for every fine grid iteration. Suppose we build a processor grid with n^p processors for every point grid with n^d gridpoints, so that all computation on the fine grid take $O(n^{d-p})$ time. Then we can show that the time for an iteration T(n), satisfies

$$T(n) = \begin{cases} O(n^{d-p}) & \text{if } c < a^{d-p} \\ O(n^{d-p} \log n) & \text{if } c = a^{d-p} \\ O(n^{\log_a c}) & \text{if } c > a^{d-p} \end{cases}$$

Thus, we get a speedup proportional to the number of processors employed only in the first case. The loss of efficiency is not too bad ($O(1/\log n)$) in the second case, as for instance when there is one processor for every gridpoint ($p = d$) and we take $c = 1$.

ACKNOWLEDGEMENT

This research was partially supported by the Office of Naval Research under contract N00014-83-K-0703. Part was done while the author was a guest researcher at the Royal Institute of Technology, Stockholm, and Uppsala University, Sweden.

14. R. Schreiber. Systolic arrays for eigenvalues. Proc. Inter-American Workshop on Numerical Methods (Caracas, 1982), editors V. Pereyra and A. Reynosa, Springer-Verlag, to appear.

15. R. Schreiber and P. Kuekes. Systolic linear algebra machines in digital signal processing. VLSI and Modern Signal Processing, editors S.Y. Kung, H.J. Whitehouse and T. Kailath, Prentice-Hall, to appear.

16. R. Schreiber. A systolic architecture for singular-value decomposition. Proc. 1er Colloque International sur les Methodes Vectorielles et Paraleles en Calcul Scientifique. Electicite de France Bulletin de la Direction des Etudes et Techerches, Serie C, No 1-1983 (1 avenue du General de Gaulle, 92140 Clamart, France.), pp.

17. R. Schreiber. On the systolic arrays of Brent and Luk for the symmetric eigenvalue and singular value problems. In preparation.

18. R. Schreiber. An improved iterative method for computing the generalized inverse of a matrix. In preparation.

19. J.J. Symanski. Progress on a systolic processor implementation. Proc. SPIE Vol 341., SPIE, Bellingham, Wash., 1982, pp. 2-7.

REFERENCES

1. J. Blackmer, P. Kuekes, and G. Frank. A 200 MOPS systolic processor. SPIE Vol 298. SPIE, Bellingham, WA, 1981, pp. 10-18.

2. A. Bojanczyk, R.P. Brent, and H.T. Kung. Numerically stable solution of dense systems of linear equations using mesh-connected processors. SIAM J. Sci. and Stat. Comput., 1983.

3. A. Brandt. Multigrid solvers on parallel computers. ICASE report 80-23, ICASE, NASA Langley Res. Ctr., Hampton, VA, 1980.

4. R.P. Brent and F.T. Luk. A systolic architecture for the singular value decomposition. Tech Rep. TR-CS-82-09 Dept. of Computer Science, Australian National University, Canberra, August, 1982.

5. R.P. Brent and F.T. Luk. A systolic architecture for almost linear time solution of the symmetric eigenvalue problem. Tech. Rep. TR-CS-82-10, Comp. Sci., ANU, 1982.

6. R.P. Brent, F.T. Luk, and C. Van Loan. Almost linear time computation of the singular-value decomposition using mesh-connected processors. Rep. TR-82-528, Dept. of Comp. Sci., Cornell Uni., Ithaca, NY, November, 1982.

7. T. Chan and R. Schreiber. Highly parallel computing networks for multigrid algorithms. In preparation.

8. S.N. Cole. Real-time computation by n-dimensional interative arrays of finite-state machines. IEEE Trans. Comp. C-18, pp. 349-365 (1969).

9. D. Gannon and J. Van Rosendale. Highly parallel multigrid solvers for elliptic PDEs: an experimental analysis. ICASE Report 82-36, ICASE, NASA Langley Research Center, Hampton, VA, 1982.

10. W.M. Gentleman and H.T. Kung. Matrix triangularization by systolic arrays. Proc. SPIE Vol 298, SPIE, Bellingham, WA, 1981, pp. 19-26.

11. D.E. Heller and I.C.F. Ipsen. Systolic networks for orthogonal equivalence transformations and their applications. Proc., Conf. on Advanced Research in VLSI, editor P. Penfield, Jr., Artech House, Dedham, MA, 1982, pp. 113-122.

12. H.T. Kung and C.E. Leiserson. Systolic arrays for VLSI, §8.3 of C.A. Mead and L.A. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, Mass., 1980.

13. R. Schreiber. Systolic arrays for eigenvalue computation. Proc. SPIE Vol 341. SPIE, Bellingham, Wash., 1982, pp. 27-34.

TRITA-NA-8311

NADA (Numerisk analys och datalogi)
KTH
100 44 Stockholm

Department of Numerical Analysis
and Computing Science
Royal Institute of Technology
S-100 44 Stockholm, Sweden

On the systolic arrays of Brent, Luk, and van Loan for the symmetric eigenvalue and singular value problems

by

Robert Schreiber

Report TRITA-NA-8311

Department of Computer Science, Stanford University CA 94305, USA
Current address: Department of Numerical Analysis and Computing Science,
Royal Institute of Technology, S-100 44 Stockholm, Sweden

ABSTRACT

Systolic architectures due to Brent, Luk, and Van Loan are today the most promising method for computing the symmetric eigenvalue and singular value decompositions in real time. These systolic arrays, however, are only able to solve problems of a given, fixed size. Here we present two modified algorithms and a modified array that do not have this disadvantage. The results of a numerical experiment show that one combination of new algorithm and array is just as efficient as the Brent-Luk-Van Loan method.

1. INTRODUCTION

Systolic arrays are of significant and growing importance in numerical computing [11], especially in matrix computation and its applications in digital signal processing [12]. There is now considerable interest in systolic computation of the singular value decomposition [2, 4, 6, 10] and the symmetric eigenvalue problem [1, 9].

To date, the most powerful systolic array for the eigenvalues of a symmetric $n \times n$ matrix is a square $n/2 \times n/2$ array due to Brent and Luk. This array implements a certain cyclic Jacobi method. It takes $O(n)$ time to perform a sweep of the method, and $O(\log n)$ sweeps for the method to converge [1].

Brent and Luk have also invented a closely related $(n/2)$-processor linear array for computing the singular value decomposition (SVD) of an $m \times n$ matrix A. SVD of A is a factorization $A = U\Sigma V^T$, where V is orthogonal, Σ is nonnegative and diagonal, and U is $m \times n$ with orthonormal columns. This array implements a cyclic Hestenes algorithm that, in real arithmetic, is an exact analogue of their Jacobi method applied to the eigenproblem for A^TA. The array requires $O(mn)$ time for a sweep, and $O(\log n)$ sweeps for convergence [2].

A new array, very like the eigenvalue array, is reported by Brent, Luk, and Van Loan to be capable of finding the SVD in time $O(m + n \log n)$ [3].

The purpose of this paper is to consider an important, indeed an essential problem concerning the practical use of these arrays. How, with an array of a given fixed size, can we solve problems of arbitrarily large size?

2. THE JACOBI AND HESTENES METHODS AND THE BRENT-LUK ARRAYS

We shall concentrate on Hestenes' method for the SVD. Starting with the given matrix A, we build an orthogonal matrix V such that AV has orthogonal columns. Thus

$$AV = U\Sigma$$

where U has orthonormal columns and Σ is nonnegative and diagonal. An SVD is given by $A = U\Sigma V^T$.

To construct V, we take

$$A^{(0)} = A,$$

and iterate

$$A^{(i+1)} = A^{(i)}Q^{(i)}, \quad i = 0, 1, \ldots$$

with $Q^{(i)}$ orthogonal until some matrix $A^{(i)}$ has orthogonal columns. $Q^{(i)}$ is chosen to be a product of $n(n-1)/2$ plane rotations

$$Q^{(i)} = \prod_{j=1}^{n(n-1)/2} Q_j^{(i)}.$$

Every possible pair (r, s), $1 \leq r < s \leq n$, is associated with one of the rotations $Q_j^{(i)}$ (the association is independent of i) in this way: the rotation $Q_j^{(i)}$ is chosen to make columns r and s of

$$A^{(i)} \prod_{k=1}^{j} Q_k^{(i)}$$

orthogonal. The process of going from $A^{(i)}$ to $A^{(i+1)}$ is called a "sweep". Every permutation of the set of pairs corresponds to a different cyclic Hestenes method.

The correspondence with the Jacobi method is this. The sequence $A^{(i)}$ converges to the diagonal matrix Σ^2 of eigenvalues of A^TA. Moreover

$$A^{(i+1)^T}A^{(i+1)} = Q^{(i)^T}(A^{(i)^T}A^{(i)})Q^{(i)}$$

where $Q^{(i)}$ is the product of $n(n-1)/2$ of Jacobi rotations that zero, in some cyclic order, each off-diagonal element of $A^{(i)^T}A^{(i)}$.

The permutation chosen by Brent and Luk allows the rotations to be applied in parallel in groups of $n/2$. Their permutation consists of $n-1$ groups of $n/2$ pairs such that, in each group, every column occurs once. Thus, the $n/2$ rotations corresponding to a pair-group commute. They can be applied in any order or, in fact, in parallel.

The SVD array is shown in Figure 1. There are $n/2$ processors. Each processor holds two matrix columns. Initially processor i holds column $2i-1$ in its "left memory" and column $2i$ in its "right memory".

In each cycle, each processor computes and applies to its two columns a plane rotation that makes them orthogonal. Next, using the connections shown in Figure 1, columns move to neighboring processors. This generates a new set of $n/2$ column-pairs.

After $n-1$ cycles, $n(n-1)/2$ pairs of columns have been generated and made orthogonal. It can be shown (by a parity argument) that no pair occurs twice during this time. Thus, every pair is generated exactly once.

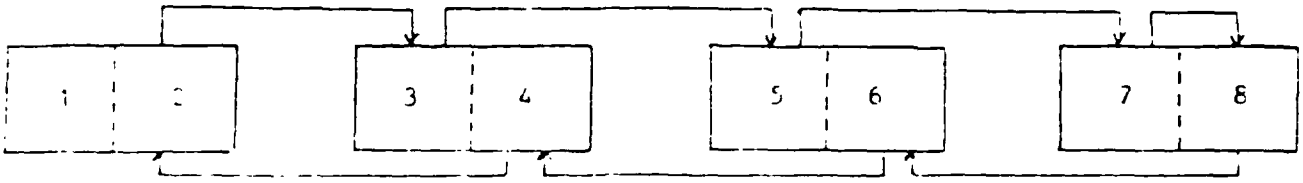


Figure 1 The Brent-Luk SVD array, n = 8

A diagram (given in [2] originally) showing the movement of columns through the array, very important in the considerations to follow, is given in Figure 2.

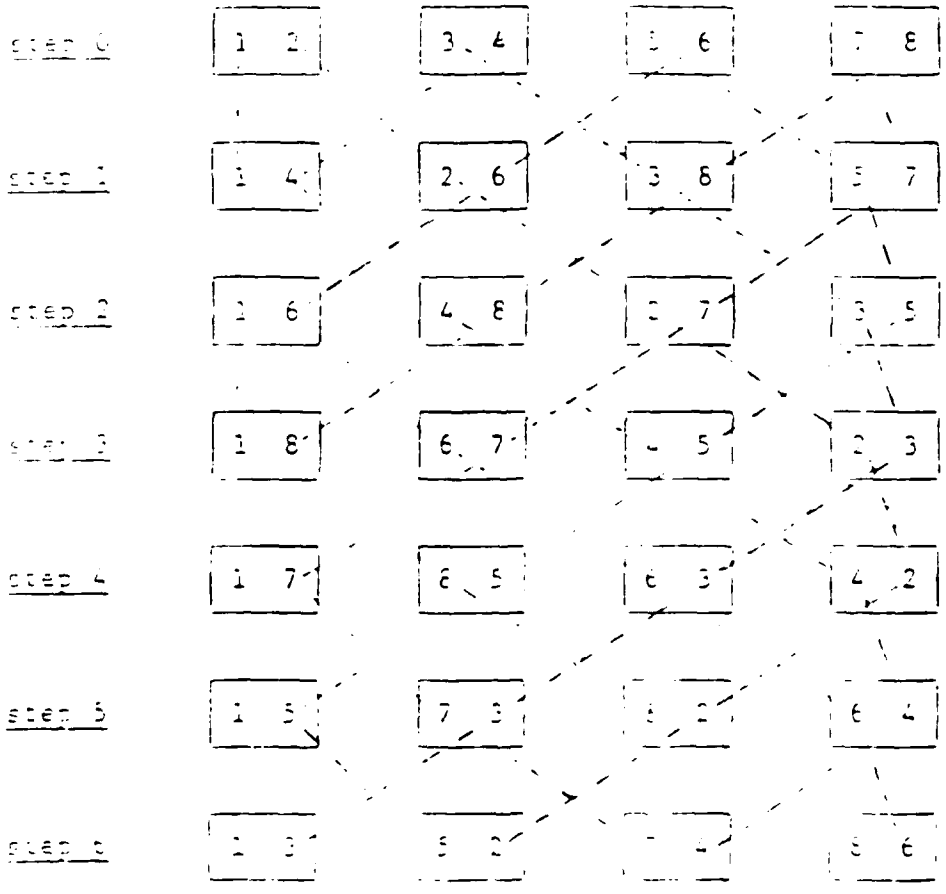


Figure 2. *Flow of data in the SVD array; n=8*

3. SOLVING LARGER PROBLEMS

We now consider the problem of finding an SVD when A has n columns, the array has p processors, and $n > 2p$.

The usual approach to this problem is to imagine that a "virtual" array, large enough to solve the problem (having $\lceil \frac{n}{2} \rceil$ or more processors,) is to be simulated by the given, small, physical array. Moreover, the simulation must be efficient. The array should not spend a large amount of time loading and unloading data.

For some arrays, this simulation is trivial. One finds a subarray of the virtual array, of the same size as the physical array, for which all the input streams are known. Clearly the action of such a subarray can be carried out and its outputs stored. These outputs then become the inputs to other subarrays. This process continues until, subarray by subarray, the computation of the entire virtual array has been performed. If this technique is possible we say that the array is "decomposable". The various matrix multiply arrays [7], the Gentleman-Kung array [5], and the Schreiber-Kuekes backsolve array [8] are good examples of decomposable arrays.

Some arrays are indecomposable: the Kung-Leiserson band-matrix LU factorization array, for example [7].

The Brent-Luk arrays are indecomposable. Consider Figure 2. Suppose a two-processor array is available. Can it simulate the four-processor array? It cannot, not efficiently anyway, since there does not exist a two-processor segment of Figure 2 for which only known data enters. If this diagram is cut by a vertical line, data flows across the line, in both directions, every cycle. The data cannot be known if only the computations on one side of the line have been performed.

Here, we shall present a solution to this problem. The idea is to have a given p-processor array simulate a pq-processor "superarray" which is not of the Brent-Luk type. Moreover, the superarray is decomposable. In its space-time dataflow graph the processors occur in groups of p. For long periods, of either p or $2p - 1$ cycles, there is no data flow between groups. Thus, the physical array can efficiently carry out the computation of the superarray, groupbygroup.

We give two such superarrays. The first implements a Hestenes method in which a "sweep" corresponds to a permutation of a multiset of offdiagonal pairs. There is some redundancy, some pairs are generated and orthogonalized several times. The second implements a cyclic Hestenes method with a permutation different from Brent and Luk's. For this method, a minor change must be made to the array.

We have compared these new sweeps to the Brent-Luk sweep. These experiments indicate that the first superarray is about $20 - 60\%$ less efficient than the Brent-Luk array, while the second superarray is virtually equal to the Brent-Luk array in efficiency.

3.1 METHOD A

The method is easiest to explain in terms of an example. Suppose we have a 4 processor array. Suppose there are 16 columns in A. We proceed as follows.

1. Load columns 1–8 and perform a Brent–Luk sweep on them;
2. Load columns 9–16 and perform a Brent–Luk sweep;
3. Load columns 1–4, 13–16; perform a Brent–Luk sweep;
4. Load columns 5–8, 9–12; perform a Brent–Luk sweep;
5. Load columns 1–4, 9–12; perform a Brent–Luk sweep;
6. Load columns 13–16, 5–8; perform a Brent–Luk sweep.

Steps 1–6 together consistute an A — supersweep or ASsweep. During an ASsweep, every column pair is generated. Some are generated more than once.

To describe the general case, suppose there is a p processor array, and $n = 2pq$ (pad A with zero columns, if necessary, so that $2p$ divides n). Imagine that the matrix A consists of q supercolumns or Scolumns: supercolumn A_i consists of columns

$$q(i-1)+1, \cdots, qi.$$

Now consider a q–superprocessor or Sprocessor virtual superarray or Sarray. Each Sprocessor holds two Scolumns (one in each of its left and right memories). In one supercycle (Scycle) the Sprocessors each perform a single Brent–Luk sweep over the $2p$ columns in their memory.

(Obviously we can simulate an Scycle of an Sprocessor using one p–processor Brent–Luk array and $2p-1$ cycles of time. Moreover, we can be loading the data for the next Scycle and unloading the data from the preceding Scycle at the same time as we process the data for the current Scycle.)

Initially, Scolumns A_1 and A_2 are in S processor 1, A_3 and A_4 in Sprocessor 2, etc.

Between Scycles, the Scolumns move to neighboring Sprocessors. The scheme for moving Scolumns is precisely the same as the scheme for moving ordinary columns in a q–processor Brent–Luk array.

After $2q-1$ Scycles, we have generated every pair of Scolumns exactly once. Together these $2q-1$ Scycles constitute an ASsweep. During an ASsweep, every pair of columns of A is orthogonalised. If two columns are in *different* Scolumns then they are orthogonalized once, during the Scycle in which their containing Scolumns occupy the same Sprocessor. If they are in the *same* Scolumn, then they are orthogonalised $2q-1$ times.

In units of cycles, the time for an ASsweep, T_{AS}, is

$$\begin{aligned} T_{AS} &= (2q-1)S\text{cycles} * (2p-1) \quad \frac{\text{cycles}}{S\text{cycle}} \\ &= (2q-1)(2p-1) \end{aligned}$$

(Of course, the simulation by a p–processor array takes q times this time.) The time for a Brent–Luk sweep over n columns, T_{BL}, is

$$T_{BL} = n - 1 = 2pq - 1.$$

Thus, the ASsweep takes longer; the ratio of times satisfies

$$\frac{9}{7} \leq \frac{T_{AS}}{T_{BL}} < 2.$$

(The lower bound arises in the simplest nontrivial case $p = q = 2$.)

There is little theoretical basis for comparing the effectiveness of ASsweeps and Brent–Luk sweeps in reducing the nonorthogonality of the columns of A. We have, therefore performed an experiment. A set of square matrices A whose elements were random and uniformly distributed in $[-1, 1]$ were generated. Both ASsweeps and Brent–Luk sweeps were used until the sum–of–squares of the off–diagonal elements of $A^T A$ was reduced to 10^{-12} times its initial value. We show the results in Table 1. The number of test matrices, the average number of sweeps, the largest number for any test matrix, and the relative time

$$\rho \equiv \frac{T_{AS} * \text{average-sweeps (AS)}}{T_{BL} * \text{average-sweeps (BL)}}$$

are shown.

Evidently one ASsweep is more effective in reducing nonorthogonality than one Brent-Luk sweep. This is not surprising, since more orthogonalisations are performed. Their cost–effectiveness, however, is roughly $20 - 60\%$ less.

				Averages		Maxima		
p	q	n	trials	AS	BL	AS	BL	ρ
2	2	8	320	3.98	4.33	5	5	1.18
2	4	16	160	5.10	5.38	6	7	1.33
2	8	32	80	6.18	6.29	7	7	1.43
4	2	16	160	4.80	5.40	5	6	1.24
4	4	32	80	5.99	6.31	7	7	1.50
4	8	64	20	7.05	7.55	8	8	1.57
8	2	32	80	5.25	6.28	6	7	1.21
8	4	64	10	6.60	7.60	7	8	1 45
16	2	64	20	6.00	7.30	6	8	1.21

Table 1. *Comparison of Break-Luk sweeps and ASsweeps*

In order to gange the reliability of the statistics generated by this experiment, we also measured the standard deviations of the sampled data. In all cases, the standard deviations were less than 0.5. For the samples of size 80 or more, the standard errors of the means are no more than 0.06, so these statistics are quite reliable. For the samples of size 20 and 10, these data may be in error by as much as 10%.

3:2 METHOD B

Method A suffers some loss of speed because, in an ASsweep, some column-pairs are generated many times. By making a small modification to the Brent-Luk array and using the new array as our basic tool, we can simulate a new supersweep, called an ABSsweep, during which every column-pair is generated exactly once.

Figure 3 shows the modified array. The connection from processor 1 to processor p is new. Note that a ring connected set of processors can easily simulate this structure. This array is still able to perform Brent-Luk sweeps over sets of $2p$ columns. But it can also perform a second type of sweep, which we call an "AB-sweep" that we now describe.

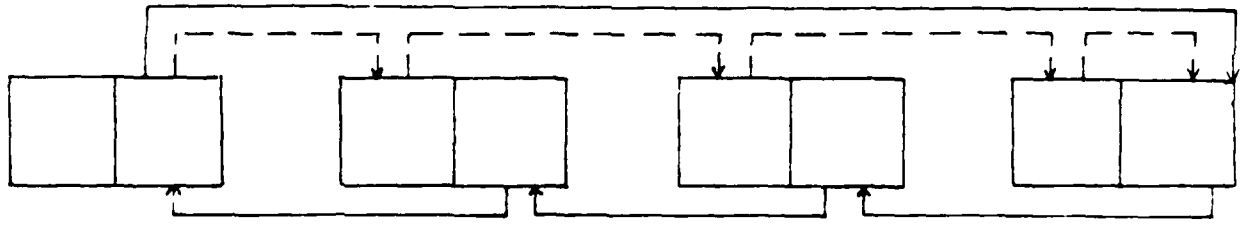


Figure 3. *The modified SVD array; n=8*

In an AB-sweep, a pair (A, B) of Scolumns, each consisting of p columns. is loaded into the array. During the sweep, all pairs $(a, b), a \in A, b \in B$ are generated exactly once. But no pairs from $A \times A$ or $B \times B$ are generated.

To implement an AB-sweep, place the columns of A in the p left memories and the columns of B in the p right memories of the processors. (The set of left (resp. right) processor memories is the Sprocessor's left (resp. right) memory, rather than the memories of the leftmost (resp. rightmost) $p/2$ processors). Processors do precisely what they did before: orthogonalize their two columns. Between cycles, A remains stationary, while B rotates one position, using the connections shown as solid lines in in Figure 3.

An ABSsweep is this. Again we work with $2q$ Scolumns of p columns each The initial configuration is as for an ASsweep. During the first Scycle, which takes $2p - 1$ cycles, every Sprocessor performs a Brent-Luk sweep on the $2p$ columns in its memory. On subsequent Scycles, all Sprocessors perform AB-sweeps, where the sets A and B are the two S columns in its memory. Between Scycles, Scolumns move as before.

It is easy to see that in an *ABS* sweep every column pair is generated once. Thus this scheme implements a true cyclic Hestenes method. The permutation differs, nevertheless, from the Brent-Luk permutation.

Again, we have compared the new scheme to the Brent-Luk scheme by an experiment. The experimental set up was precisely the same as for the previous experiment. The results are shown in Table 2.

				Averages		Maxima	
p	q	n	trials	ABS	BL	ABS	BL
2	2	8	320	4.32	4.33	5	5
2	4	16	160	5.35	5.38	6	7
2	8	32	80	6.36	6.29	7	7
4	2	16	160	5.36	5.40	6	6
4	4	32	80	6.18	6.31	7	7
4	8	64	20	7.50	7.55	8	8
8	2	32	80	6.13	6.28	7	7
8	4	64	10	7.10	7.60	8	8
16	2	64	20	7.00	7.30	7	8

Table 2. *Comparison of Brent-Luk sweeps and ABS sweeps*

Evidently, *ABS* sweeps are as effective as Brent-Luk sweeps. The standard deviations of the number of *ABS* sweeps needed were also all less than 0.5.

4. THE EIGENVALUE ARRAY

Decomposition of the eigenvalue array presents the same difficulty, and is amenable to the same solution, as the *SVD* array. We need not present the details here. Note, however, that in simulating a $pq \times pq$ eigenvalue array, a $p \times p$ array must be used to simulate both diagonal subarrays where its diagonal processors generate rotations, and off-diagonal subarrays, where all its cells only apply rotations. The array of Brent, Luk, and Van Loan for the *SVD* can also be treated in this way.

ACKNOWLEDGEMENT

This work was done in Stockholm while I was a guest of the Royal Institute of Technology, Department of Numerical Analysis and Computer Science, and of Uppsala University, Department of Computer Science. The research was also partially supported by the U.S. Office of Naval Research under contract N00014-82-K-0703.

The work's inspiration came from discussions with Erik Tidén and Björn Lisper, whom I thank.

REFERENCES

1. R.P. Brent and F.T. Luk. *A systolic architecture for almost linear-time solution of the symmetric eigenvalue problem.* Tech. Rept. TR-82-525, Computer Science Dept. Cornell Univ. Aug., 1982.
2. R.P. Brent and F.T. Luk. *A systolic architecture for the singular value decomposition.* Tech. Rept. TR-82-522, Computer Science Dept., Cornell Univ., Sept., 1982.
3. R.P. Brent, F.T. Luk and C. Van Loan. *Computation of the singular value decomposition using mesh-connected processors.* Tech.Rept. TR-82-528, Computer Science Dept., Cornell Univ., March 1983.
4. A.M. Finn, F.T. Luk and C. Pottle. *Systolic array computation of the singular value decomposition.* Proc. 1982 SPIE Tech. Symp. East, Vol 341, *Real-Time Signal Processing V.* Society of Photo-optical Instrumentation Engineers, Bellingham, Wash. (1982).
5. W.M. Gentleman and H.T. Kung. *Matrix triangularization by systolic array.* Proc. 1981 SPIE Tech. Symp. Vol 298, *Real-Time Signal Processing IV.* Society of Photo-optical Instrumentation Engineers, Bellingham, Wash. (1981).
6. D.E. Heller and I.C.F Ipsen. *Systolic networks for orthogonal decompositions.* SIAM J. Scient. and Stat. Comp., June 1983.
7. H.T. Kung and C.E. Leiserson. *Systolic arrays for (VLSI).* Section 8.3 of C. Mead and L. Conway, *Introduction to VLSI Systems,* Addison-Wesley, 1980.
8. R. Schreiber and P.T. Kuekes. *Systolic linear algebra machines in digital signal processing.* Proc. USC Workshop on VLSI and Modern Signal Processing, Prentice-Hall, to appear.
9. R. Schreiber. *Systolic arrays for eigenvalue computation.* Proc. 1982 SPIE Tech Symp. East, Vol. 341, *Real-Time Signal Processing V.* Society of Photo-optical Instrumentation Engineers, Bellingham, Wash. (1982).
10. R. Schreiber. *A systolic architecture for singular value decomposition.* Proc. 1er Colloque International sur les Méthodes Vectorielles et Paralèles en Calcul Scientifique. Electricité de France, Bulletin de la Direction des Etudes et Recherches, Série C, No 1-1983.
11. R. Schreiber. *Systolic arrays: High performance parallel machines for matrix computation.* Elliptic Problem Solvers, Garret Birkhoff and Arthur Schoenstadt, eds., Academic Press, to appear.
12. J.M. Speiser and H.J. Whitehouse. *Architectures for real time matrix operations.* Proc. Government Microcircuits Applications Conference, Houston, 1980.

KTH
Inst för Numerisk analys och datalogi
100 44 Stockholm

Dept of Numerical Analysis
and Computing Science
The Royal Institute of Technology
S-100 44 Stockholm, Sweden

ON SYSTOLIC ARRAY METHODS FOR BAND MATRIX FACTORIZATIONS

by

Robert Schreiber *

TRITA-NA-8316

* Department of Numerical Analysis and Computing Science
The Royal Institute of Technology, Stockholm, Sweden

Permanent address: Computer Science Department, Stanford University,
Stanford CA 94305, USA

1. INTRODUCTION

In this paper we present a new systolic array for band matrix QR factorization. We then point out the relationships between this array and three other systolic arrays: the hexagonally connected LU factorization array of Kung and Leiserson [4], the Heller-Ipsen array for QR factorization [3], and an LU variant of the Heller-Ipsen array that has not actually appeared anywhere in the literature.

Next, we show how to compute the QR factorization of a banded rectangular matrix on a systolic array. Matrices of this kind arise in least-squares collocation methods for elliptic differential equations and some integral equations.

The results presented here are also applicable to the solution of ill-posed problems by regularization methods. The details of this application shall appear in another paper [2]. In fact, the application was investigated first, by Eldén.

1.1 Preliminary concepts

Let B be any matrix. We say B is a (p,q) banded matrix if

$$b_{ij} = 0 \quad \text{for all} \quad i > j + p \quad \text{and all} \quad j > i + q.$$

We let $w \equiv 1 + p + q$, the number of nonzero diagonals of B. When necessary, we write p(B) (q(B),w(B)) to distinguish the matrix in question.

We are concerned with the computation of factorizations

$$B = QR \tag{1.1}$$

where Q is orthogonal and R upper triangular and also factorizations

$$PB = LR \tag{1.2}$$

where P is a permutation matrix, L is lower triangular with diagonal elements one, and R is upper triangular. The factors L and R are also banded, with

$$q(U) \leq \min(n-1,\ p(B) + q(B))$$
$$p(L) = p(B)$$

We let $\underline{b}_i$ denote the i^{th} row of B, $1 \leq i \leq n$.

We shall consider the implementation by systolic arrays of two algorithms for the factorizations (1.1) and (1.2). The algorithms are alike in that they zero every element below the main diagonal of B, in predetermined sequence. The QR algorithms use plane rotations to do so; the LR algorithms use elementary row operations. What distinguishes the two QR algorithms is the pair of rows rotated in zeroing an element; say b_{ij}. In one of them, rows i-1 and i are used; in the other rows j and i are used. For the LR algorithms the distinction is the same: to zero b_{ij} one subtracts a multiple of either row i-1 or row j from row i. To be specific, we use either

Algorithm 1 (Central strategy):

```
for j := 1 to n-1
    for i := j+1 to min(n,j+p)
        process (b_i, b_j, j)
```

or

Algorithm 2 (Neighbor strategy)

```
for j := 1 to n-1
    for i := min(n,j+p) to j+1 step -1
        process (b_i, b_{i-1}, j)
```

In the case of a QR factorization, process $(\underline{x},\underline{y},j)$ zeros the j^{th} element of $\underline{x}$ by applying a plane rotation to $\underline{x}$ and $\underline{y}$. In the case of an LR factorization, process $(\underline{x},\underline{y},j)$ zeros the j^{th} element of $\underline{x}$ by subtracting a multiple of $\underline{y}$ from $\underline{x}$, after first exchanging $\underline{x}$ and $\underline{y}$, if necessary, to keep the absolute value of the multiplier less than or equal to 1.

For some matrices, the LR factorization exists and can be accurately computed by the central strategy without row interchanges. This is the algorithm implemented by Kung and Leiserson [4].

In discussing parallel processor arrays we use terms and conventions that have become standard; one can consult the review paper of Brent, Kung, and Luk [1] for background. In particular, we use the term efficiency to denote the fraction of processor cycles that a typical processor is actively employed in an array.

2. A NEW ARRAY FOR BAND MATRIX QR FACTORIZATION

We present here an array implementing Algorithm 1 for QR factorization. The array, and the manner in which imput data enters, is shown in Figure 1; the format of the output in Figure 2; the definitions of the cells in Figure 3.

The efficiency of this array is 1/3.

Three identically structured matrices could be interleaved and factored simultaneously. In that case the efficiency would be 1.

The array uses $(p+1)(q + p/2 + 1)$ cells of which $q+1$ are simply delay cells (diamonds in Figure 1), p generate rotations (circles in Figure 1), and the rest apply rotations (squares in Figure 1).

If b_{11} enters the array at time 1, then b_{nn} enters at time $1 + 3(n-1)$; the last output element, r_{nn}, leaves the array at time $3n + 2p - 1$.

In its interconnection structure this array is the same as the Kung-Leiserson array [4], but it has more cells, of different types.

By making obvious changes to the circle and square cells, we can create an LR factorization array. Both these arrays use the central strategy of Algorithm 1.

An array for QR factorization using the neighbor strategy of Algorithm 2 was given by Heller and Ipsen [3]. With a similar change to the cell definitions their array becomes an LR factorization array, too. The Heller-Ipsen array contains pw active cells and is 1/2-efficient. It has no delay cells. The last output element emerges at time $2n + 2p - 1$, so it is faster than the present array.

The principle advantage of the present array is its applicability to generalizations of the problem for which it was devised. We illustrate one such application in the next sections, and another in the paper mentioned earlier [2].

→ carries plane rotations;

carry matrix elements.

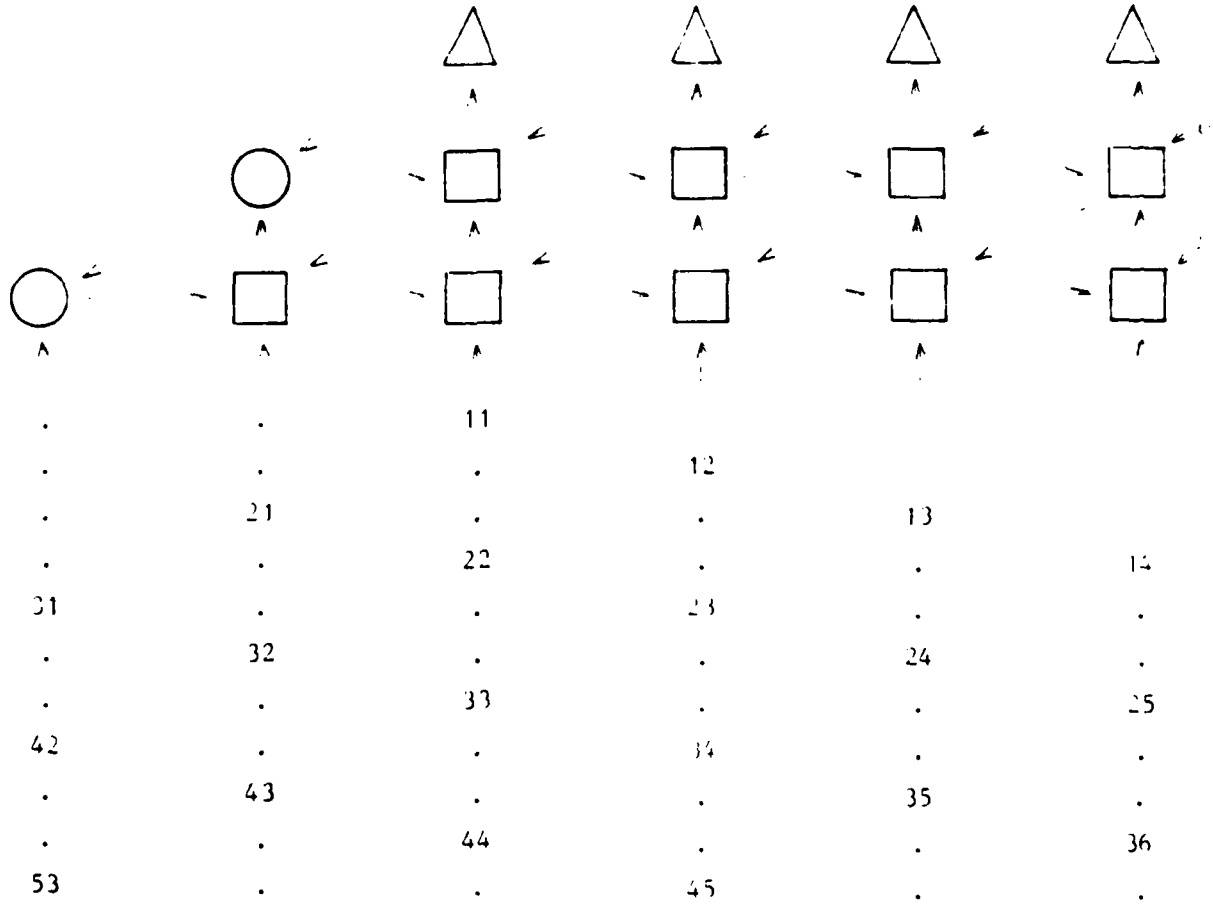


Figure 1. The array for QR and LR factorization by the central strategy

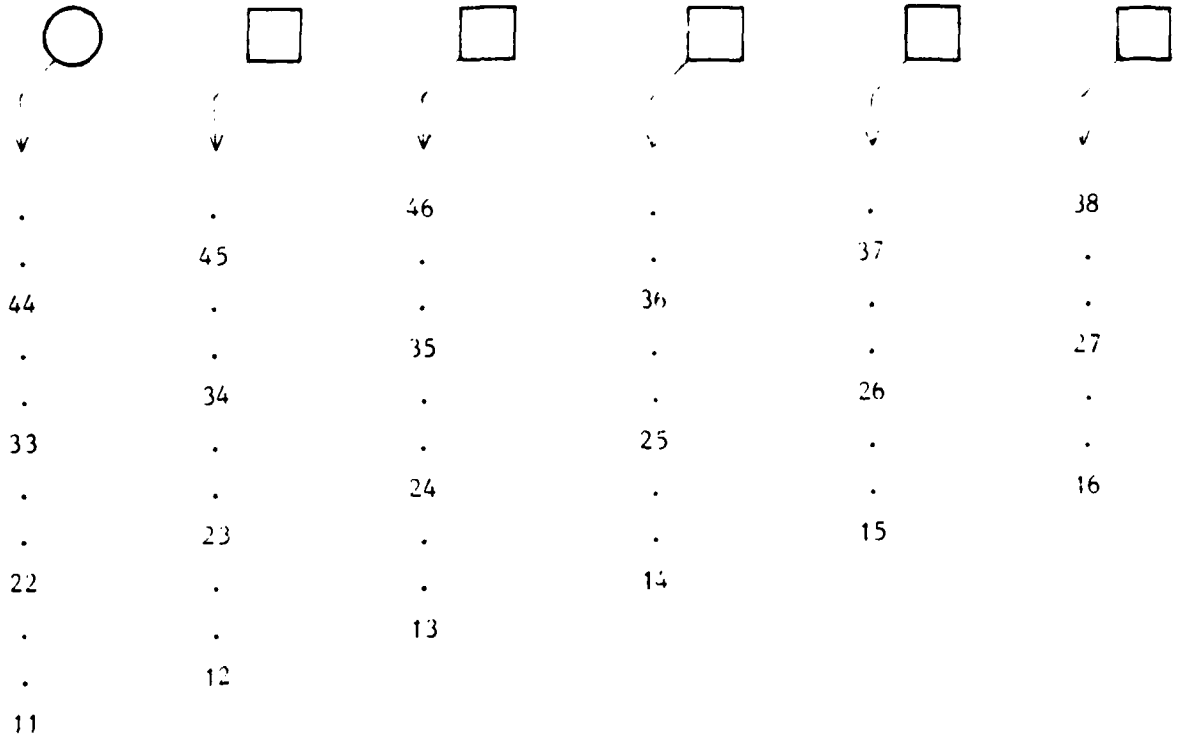
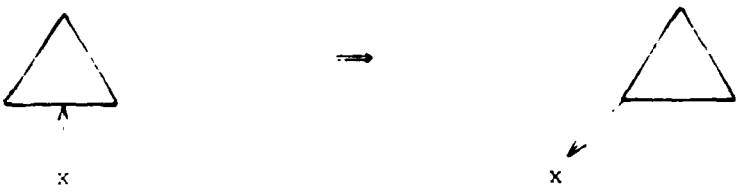


Figure 2. Output from the array

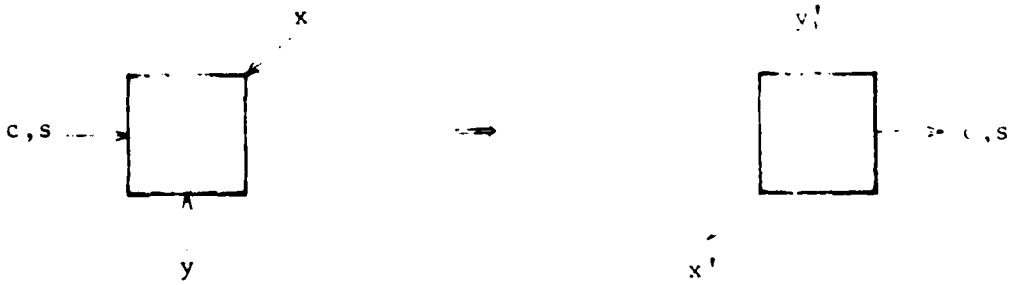
delay cell:



boundary cell:

where $c = \cos\theta$, $s = \sin\theta$, $\begin{pmatrix} x' \\ 0 \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$

internal cell:



where $\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$

Figure 3. Cells for the QR array

3. RELATED ARRAYS

In this section we present two arrays similar to the band QR array. The first computes QR factorizations of lower triangular, banded, rectangular matrices. The second, which has a rather unusual structure in which two triangular arrays are interleaved, computes QR factorizations of block 2×1 matrices

$$\begin{bmatrix} A \\ B \end{bmatrix}$$

in which both A and B are lower triangular, banded, rectangular matrices. In the next section, this array is used to generate QR factorizationsof banded rectangular matrices of a kind arising from integral and differential equations. With the obvious changes to the cell definitions, all the results apply to LR factorization, too

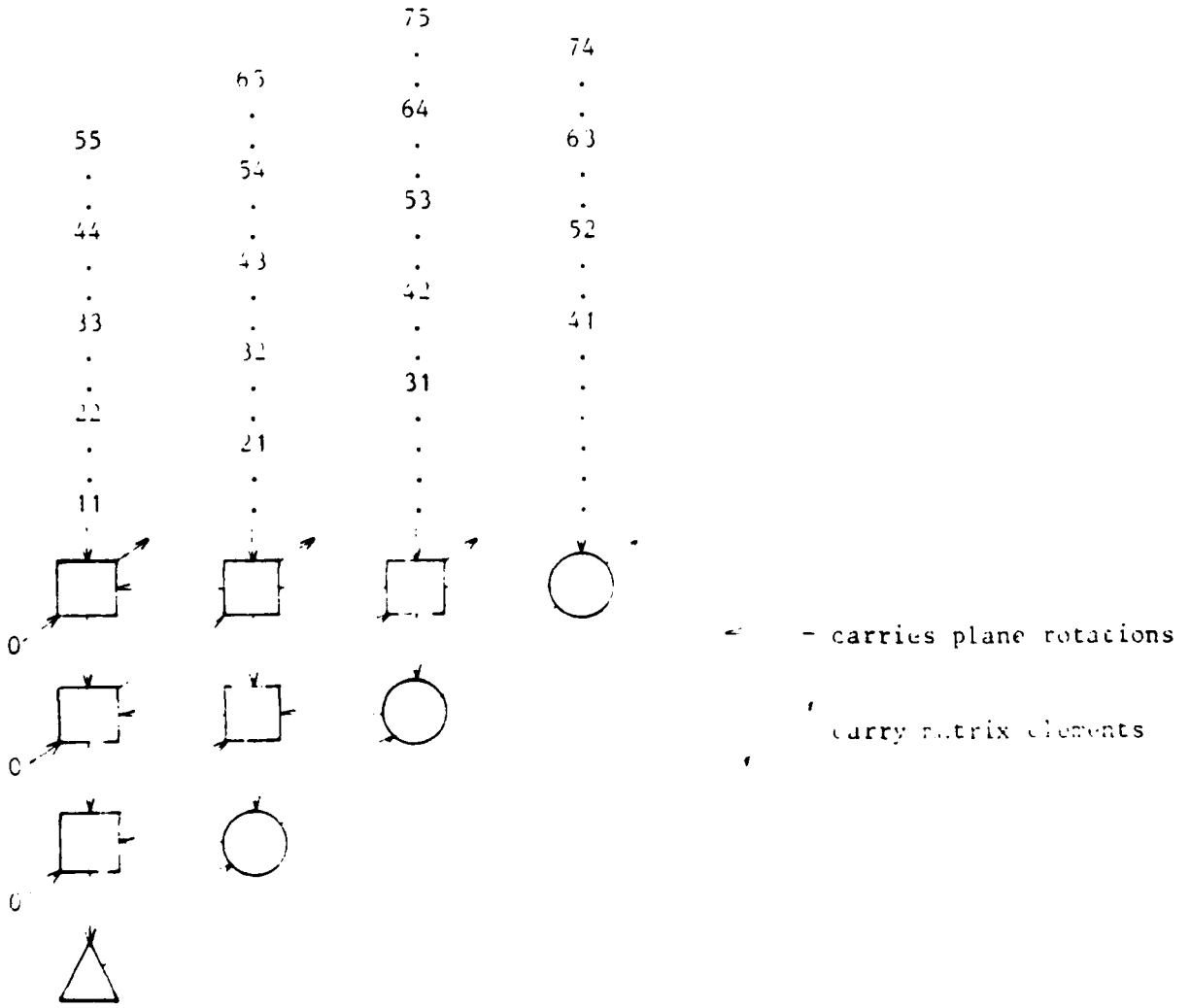


Figure 4. QR factorization of a BLTE (3,2) matrix with 5 columns

3.1 Factorization of lower triangular, banded elongated matrices

Definition. A matrix L is banded (p), lower triangular, elongate (s) (BLTE (p,s)) if

1. L is $n+s \times n$,
2. $\ell_{ij} = 0$ if $i < j$ or $j < i-p$.

By 2, for any row $i > n + p$, every element $\ell_{ij} = 0$; thus we may assume that $0 \leq s \leq p$. For example, if L is BLTE (3,2) then

$$L = \begin{bmatrix} x & & \\ x & & 0 \\ x & & \\ x & & \ddots \\ & & \ddots \ x \\ 0 & & \ddots \ x \\ & & x \ x \end{bmatrix}$$

In Figure 4, we illustrate a systolic array for QR factorization of a BLTE (p,s) matrix where p = 3 and s = 2. In general the array is $(p+1)\times(p+1)$ and triangular with horizontal, vertical, and diagonal connections. In fact, it is the array of Section 2, specialized to the case q = 0.

Note that the input format is the same as for a square banded matrix, although its extent in time-space is different. Efficiency is 1/3. The output, an n×n square, upper triangular (o,p) banded matrix, emerges from the top cells in the format shown in Figure 2.

3.2 Factorization of block 2×1 matrices with BLTE blocks

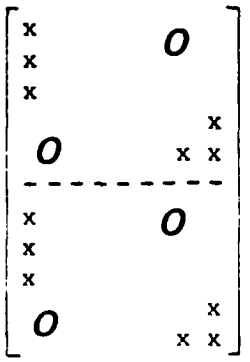
Here we consider factorization of matrices

$$\begin{bmatrix} A \\ B \end{bmatrix}$$

where A is BLTE (p_A, s_A), B is BLTE (p_B, s_B), and

$$p_B = p_A \quad \text{or} \quad p_A - 1$$

Before presenting the array, we sketch the algorithm. Let us take a specific case, $p_A = p_B = 2$, $s_A = s_B = 1$, for illustration. We must factor a matrix whose form is



The strategy is this. We eliminate elements of A in the lower triangle by a central strategy. There are two such elements in a typical column. Let them be eliminated at times 2 and 4. There are three elements to be eliminated in the corresponding column, say column j, of B. Let these be eliminated, by rotations involving row j of A, at times 1, 3, and 5.

This strategy causes no unnecessary nonzero elements to be created in either A or B. When eliminating $b_{j+r,j}$ $(0 \le r \le s_B)$ row j+r of B has nonzeros in columns j, j+1, ... ,j+r. At the same time, row j of A has precisely the same structure. So there is no "fill-in", no new nonzero is created. When $a_{j+r,j}$ $(1 \le r \le s_A)$ goes, row j gets a nonzero element in column j+r. These are the only fill-ins.

The array is shown in Figure 5. It consists of two interleaved triangular arrays. One of these, the A array, is just the array of Section 3.1. It includes those cells in odd-numbered columns of the array. They operate only on elements of A, reducing it to an upper triangular $(0, p_A)$ banded matrix. In the other array, which occupies the even-numbered columns, plane rotations are applied to pairs of rows, a row from A and a row from B, to zero all the elements of B.

In Figure 5 elements of A are shown by their indices, elements of B are shown as b_{ij}. S_{ij} (R_{ij}) represents the rotation that has zeroed b_{ij} (a_{ij}). The case $p_A = p_B = 3$ is shown. If p_B were 2, we would eliminate the top row of the B-array. If $p_B < p_A - 1$ (or $p_B > p_A$) we must include zero diagonals in the band of B (or A) to bring p_B up to $p_A - 1$ (or p_A up to p_B), otherwise the array will not work correctly.

The efficiency is 1/4. Cells active at the time shown are emphasized. The circled indices in Figure 5 represent elements of the upper triangle of A. When leaving the array at the top, they are the elements of R.

4. FACTORIZATION OF BANDED RECTANGULAR MATRICES

In least squares collocation method for ordinary and elliptic boundary value problems or certain integral equations, for example

$$\int_0^1 k(x,y)u(y)dy = g(x)$$

where $k(x,y) = 0$ if $|x-y| \le d < 1$, we may encounter rectangular banded matrices. Thus we consider QR factorization of $m \times n$ matrices A where, for some integer $\rho \ge 1$,

$$m = 1 + \rho(n-1). \tag{4.1}$$

To generalize the notion of a diagonal, we define, for $1 \le j \le n$

$$r(j) \equiv 1 + \rho(j-1)$$

and consider the elements $a_{r(j),j}$ to be the main diagonal. Then we let

$$c(i) \equiv r^{-1}(i) = 1 + [(i-1)/\rho].$$

We say that A is (p,q) banded if

$$a_{ij} = 0 \text{ unless } c(i) - p \le j \le c(i) + q. \tag{4.2}$$

For example, when $\rho = 3$, $p = q = 1$, and $n = 4$, A has the form

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\left[\begin{array}{cccc}
x & x & & \\
x & x & & \\
x & x & & \\
x & x & x & \\
 & x & x & \\
 & x & x & \\
 & x & x & x \\
 & & x & x \\
 & & x & x \\
 & & x & x
\end{array}\right]
\end{array}$$

The first step in the systolic factorization reorders the rows of A, so that after reording,

$$A = \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{bmatrix}$$

and each A_i is banded. A_0 consists of rows $1, 1+p, 1+2p, \ldots, m$ (rows 1, 4, 7, and 10 in the example shown) and is (p,q) banded. For $1 \le i < p$, A_i contains rows $i+1, i+1+p, \ldots, i+1+p(n-2)$; it is (p-1,q) banded, and it is $n-1 \times n$.

Next, we perform $p-1$ QR factorizations; first

$$\begin{bmatrix} A_0 \\ A_1 \end{bmatrix} = Q_1 R_1 \tag{4.3}$$

where R_1 is $n \times n$, then, for $2 \le i < p$

$$\begin{bmatrix} R_{i-1} \\ A_i \end{bmatrix} = Q_i R_i \,. \tag{4.4}$$

R_{p-1} is the desired factor of A; the rotations that constitute Q are the rotations that consitute Q_i, $1 \le i < p$. (We shall denote elements of R_i by $r_{k\ell}^{(i)}$.)

To compute the factorizations (4.3) and (4.4) we use the array of Section 3.2. It is first necessary to put the matrices into the form required: block 2×1 in which both blocks are BLTE. To do so, we add zero rows. For (4.3), we add

q zero rows to A_0 and

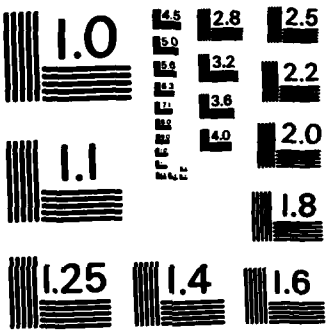
q zero rows to A_1, so that

A_0 becomes BLTE (p+q,q) and

A_1 becomes BLTE (p+q-1,q-1). Thus the factorization can be done by a $(p+q+1) \times (p+q+1)$ triangular A array with an embedded B array of size $(p+q) \times (p+q)$.

AD-A137 624 NUMERICAL METHODS FOR PARTIAL DIFFERENTIAL EQUATIONS 2/2
(U) STANFORD UNIV CA DEPT OF COMPUTER SCIEECE
R SCHREIBER 09 JAN 84 N00014-82-K-0703

UNCLASSIFIED F/G 12/1 NL



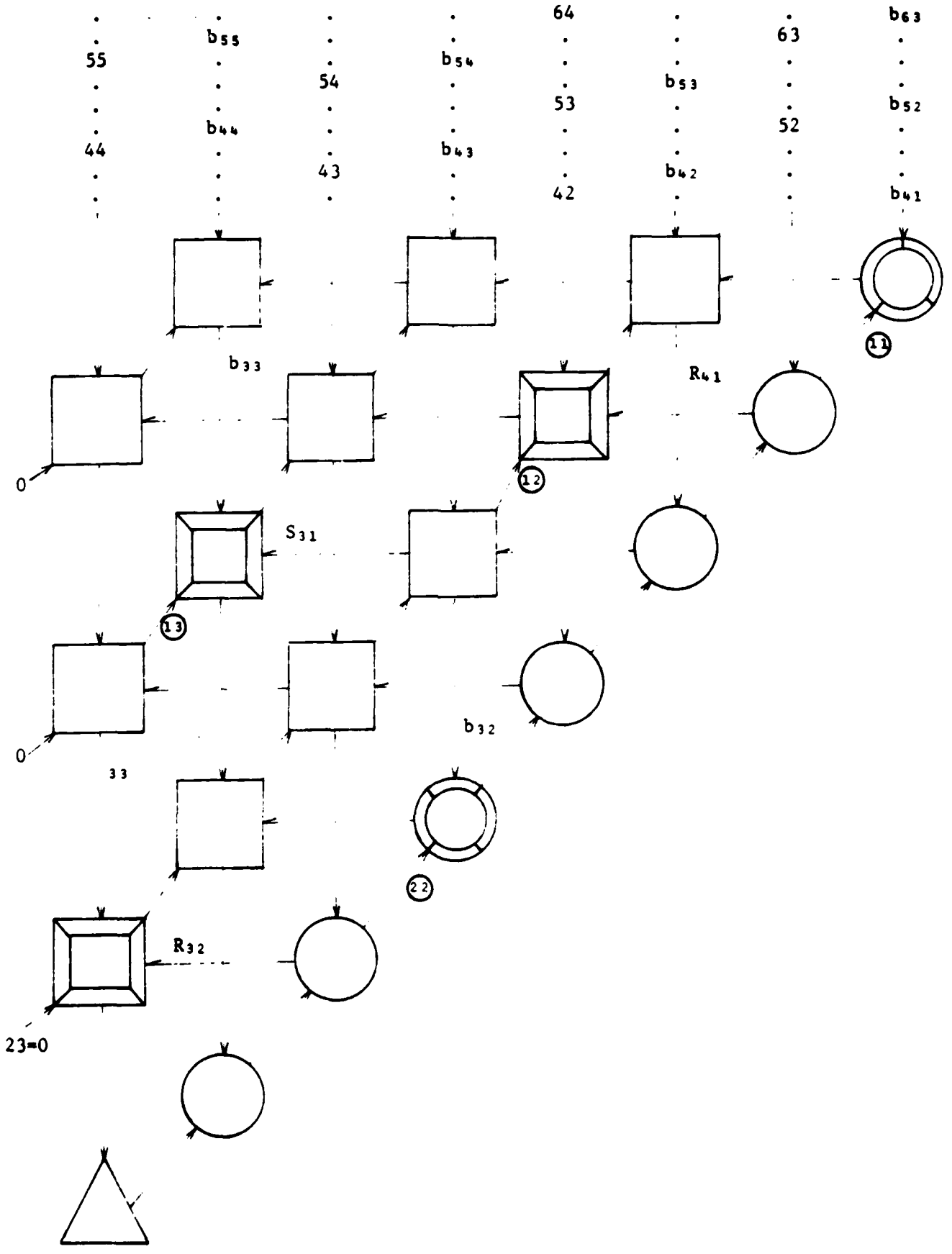


Figure 5. The $\begin{bmatrix} A \\ B \end{bmatrix}$ array

For (4.4), we note that R_{i-1} is (0, p+q) banded. So we include p+q zero rows on top to make it BLTE (p+q,p+q). A_i is treated like A_1. The same array can handle the work.

Because elements of the matrices enter every 4 clocks, it takes $4n + 3(p+q) + O(1)$ clocks to complete any of the factorizations (4.3) and (4.4); 3(p+q) is the length of the longest path through the array. The overall factorization need not take $\rho-1$ times as long, since the factorizations (4.4) can be performed in a pipelined manner. In fact, if we say that a factorization (4.4) begins when $r_{11}^{(i-1)}$ enters the array (it is the first element of R_{i-1} to do so) then we can begin the next such factorization $1 + p + q$ clocks later, as this is when $r_{11}^{(2)}$ leaves. Thus, if enough hardware is available, the entire factorization can be obtained in $4n + (\rho+2)(p+q) + O(1)$ clocks.

To achieve this throughout, we need hardware to work on roughly

$$\min(\rho,\ 4n/(p+q))$$

factorizations at a time, the second term being the ratio of the time for a factorization to the interval at which factorizations can begin. A single array can work on 4 (interleaved) factorizations at once. Thus, $\min(\rho/4,\ n/(p+q))$ arrays of $(p+q)^2$ cells can be used. These designs are fully efficient when 4 or more factorizations are simultanously performed.

ACKNOWLEDGEMENT.

I thank Lars Eldén for suggesting this research and for his helpful comments.

REFERENCES

[1] R.P. Brent, H.T. Kung, and F.T. Luk, *Some linear-time algorithms for systolic arrays*, Cornell University Department of Computer Science, Technical Report TR 83-541 (1982). Also, 9th World Computer Congress, Paris, September 1983.

[2] Lars Eldén and Robert Schreiber, *An application of systolic arrays to linear discrete ill-posed problems*. Technical Report Linköping University, Dept of Mathematics. In preparation.

[3] Don E. Heller and Ilse C.F. Ipsen, *Systolic networks for orthogonal decompositions*, SIAM Jour. Scient. and Stat. Comput. 4, pp. 261-269 (1983).

[4] H.T. Kung and C.E. Leiserson, *Systolic arrays (for VLSI)*. In C.A. Mead and L.A. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980.

On the systolic arrays of Brent, Luk, and Van Loan

Robert Schreiber

Computer Science Department, Stanford University, Stanford, California 94305

Abstract

Systolic architectures due to Brent, Luk, and Van Loan are today the most promising idea for computing the symmetric eigenvalue and singular value decompositions in real time. These systolic arrays, however, are only able to solve problems of a given, fixed size. Here we present two modified algorithms and a modified array that do not have this disadvantage. The results of a numerical experiment show that a combination of one of the new algorithms and the new array is just as efficient as the Brent-Luk-Van Loan method.

Introduction

Systolic arrays are of significant and growing importance in numerical computing, especially in matrix computation and its applications to digital signal processing.[1,2] There is now considerable interest in systolic computation of the singular value decomposition[3,4,5,6] and the symmetric eigenvalue problem.[7,8] These decompositions are needed to implement some recently advocated methods in signal processing.[9,10]

To date, the most powerful systolic array for the eigenvalues of a symmetric n x n matrix is a square n/2 x n/2 array due to Brent and Luk. This array implements a certain cyclic Jacobi method. It take O(n) time to perform a sweep of the method and O(log n) sweeps for the method to converge.[7] A very similar array is used by Brent, Luk, and Van Loan to compute the singular value decomposition (SVD) of an m x n matrix in time m + O(n log n).[11] Brent and Luk had previously described an n/2-processor linear array that requires O(mn log n) time for the SVD.[3]

This paper deals with the important practical issue of how, with an array of a given fixed size, we can solve problems of arbitrary size.

The Jacobi and Hestenes methods and the Brent-Luk arrays

We shall concentrate on Hestenes' method for the SVD. Let A be a given m x n matrix. An SVD of A is a factorization $A = U\Sigma V^t$ where V is orthogonal, Σ is nonnegative and diagonal, and U has orthonormal columns. The method starts with the given matrix A and builds an orthogonal matrix V such that AV has orthogonal columns. An SVD is obtained by normalizing the columns of AV.

The orthogonal transformation V consists of a sequence of plane rotations. Each rotation orthogonalizes a pair of columns of A. Column pairs are orthogonalized one at a time until all pairs have been orthogonalized. This constitutes one sweep of the method. The order in which pairs are orthogonalized is fixed. The method is equivalent, in real arithmetic, to a cyclic Jacobi method for the eigenvalues of A^tA.

The order chosen by Brent and Luk allows the rotations to be applied, in parallel, in groups of n/2. The linear array that does this is shown in Figure 1. There are n/2 processors. Each processor holds two matrix columns. Initially processor i holds column 2i-1 in its "left" memory and column 2i in its "right" memory. In every cycle, every processor applies a plane rotation to the columns in its two memories. The processors choose these plane rotations to make the resulting columns orthgonal. Then, columns move to adjacent processors using the connections shown in Figure 1. This generates a new set of n/2 pairs of columns.

After n-1 cycles, n(n-1)/2 pairs of columns have been generated and made orthogonal. It can be shown that no pair occurs twice. Thus every pair is generated exactly once.

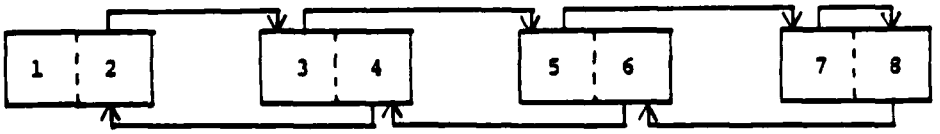


Figure 1. The Brent-Luk SVD array; n=8.

Figure 2 gives a diagram showing the movement of columns through the array.

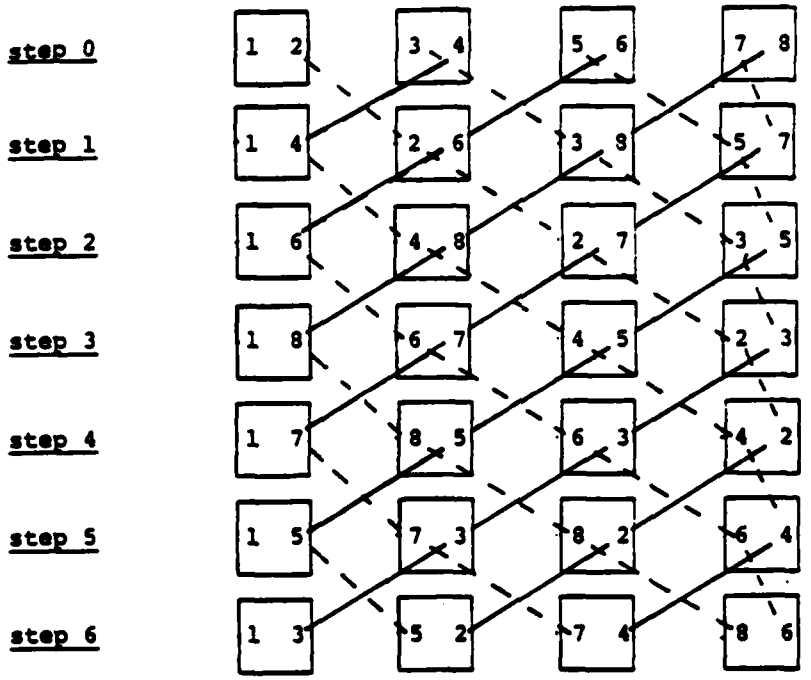


Figure 2. Flow of data in the SVD array; n = 8

Solving larger problems

We now consider the problem of finding the SVD of A when A has n columns, the array has p processors, and $n > 2p$. The usual approach to this problem is to imagine that a "virtual" array large enough to solve the problem (which means, in this case, that the array has at least n/2 processors) is to be simulated by the smaller array which is actually available. Moreover, the simulation must be efficient. The array should not spend a lot of time loading and unloading data.

For some arrays this simulation is trivial. One finds a subarray of the virtual array that is the same size as the physical array and for which all the input streams are known. The computation of such a subarray can be carried out and its outputs stored. These outputs then become the inputs to other subarrays. This process continues until, subarray by subarray, the computation of the entire virtual array has been performed. If this technique is possible, we say that the array is "decomposable". The various matrix multiplication arrays,[12] the Gentleman-Kung array,[13] and the Schreiber-Kuekes backsolve array[14] are good examples of decomposable arrays. Some arrays are indecomposable: the Kung-Leiserson band matrix LU factorization array,[12] for example.

The Brent-Luk arrays are indecomposable. Consider Figure 2. Suppose a two-processor array is available. Can it simulate a four-processor array? It cannot, not efficiently, anyway, since there does not exist a two-processor segment of Figure 2 for which only known data enters. If this diagram is cut by a vertical line, data flows across the line, in both directions, every cycle. The entering data cannot be supplied if only the computation on one side of the line is performed.

We shall present a solution to this problem. The idea is to have the p-processor array simulate a pq-processor "superarray" which is not of the Brent-Luk type. The superarray is decomposable into p-processor arrays. If one were to make a diagram like that of Figure 2, showing the flow of data in space and time in this superarray, the processors would occur in groups of p, and for rather long periods there would be no data flowing between the groups. Thus the physical array can efficiently carray out the computation of the superarray, group by group.

We give two such superarrays. The first implements a method in which, during a sweep, some pairs of columns are orthogonalized more than once. The second implements a cyclic Hestenes method, but the order in which pairs are orthogonalized is different from the

order used by Brent and Luk. In order to implement this method, a slight modification of the Brent-Luk array is required.

We have compared these new sweep orders to the one used by Brent and Luk. These experiments indicate that the first superarray is less efficient, by between 20% and 60%, than the Brent-Luk array, while the second is as efficient as the Brent-Luk array.

Method A

The method is easiest to explain by giving an example. Suppose we have a 4-processor array. Suppose there are 16 columns in A. We proceed as follows:

1. Load columns 1-8 and perform a Brent Luk sweep;
2. Load columns 9-16 and perform a Brent-Luk sweep;
3. Load columns 1-4, 13-16; perform a Brent-Luk sweep;
4. Load columns 5-8, 9-12; perform a Brent-Luk sweep;
5. Load columns 1-4, 9-12; perform a Brent-Luk sweep;
6. Load columns 13-16, 5-8; perform a Brent-Luk sweep.

Steps 1-6 together constitute an A-supersweep or ASsweep. During an ASsweep every column pair is generated. Some are generated more than once.

To describe the general case, suppose there is a p processor array and that n = 2pq. (Pad A with zero columns, if necessary, so that 2p divides n.) Imagine that the matrix A consists of q supercolumns or Scolumns. Scolumn A_i consists of columns q(i-1)+1, . . ., qi of A. Now consider a q-superprocessor (or Sprocessor) virtual superarray (or Sarray). Each Sprocessor holds two Scolumns, one in its left and one in its right memory. In one supercycle (or Scycle) each Sprocessor performs a single Brent-Luk sweep over the 2p columns in its memories. Obviously a p-processor Brent-Luk array can implement an Scycle of an Sprocessor during 2p-1 of its own cycles. Moreover, it can load data for the next Scycle and unload data from the preceding Scycle at the same time.

Initially Scolumns A_1 and A_2 are in Sprocessor 1, A_3 and A_4 in Sprocessor 2, etc. Between Scycles, the Scolumns move to neighboring Sprocessors. The scheme for moving Scolumns is precisely the same as the scheme for moving ordinary columns in a q-processor Brent-Luk array.

After 2q-1 Scycles, every pair of Scolumns has been generated exactly once. Together these 2q-1 Scycles constitute an ASsweep. During an ASsweep, every pair of columns of A is orthogonalized. If two columns are in different Scolumns, then they are orthogonalized once, during the Scycle in which their containing Scolumns occupy the same Sprocessor. If they are in the same Scolumn, then they are orthogonalized 2q-1 times.

In units of ordinary processor cycles, the time for an ASsweep, T_{AS}, is

$$T_{AS} = (2q-1)(2p-1) \text{ cycles.}$$

The time for a Brent-Luk sweep over n columns, T_{BL}, is

$$T_{BL} = n-1 = 2pq-1.$$

Thus, the ASsweep takes longer; the ration of times satisfies

$$\frac{9}{7} \leq \frac{T_{AS}}{T_{BL}} \leq 2.$$

(The lower bound arises in the simplest nontrivial case, p = q = 2.)

There is little theoretical basis for comparing the effectiveness of ASsweeps and Brent-Luk sweeps in reducing the nonorthogonality of columns of A. We have, therefore, performed an experiment. A set of square matrices A whose elements were random and uniformly distributed in [-1,1] were generated. Both ASsweeps and Brent-Luk sweeps were used until the sum of the squares of the off-diagonal elements of A^tA was reduced to 10^{-12} times its original value. We show the results in Table 1. The number of test matrices, the average number of sweeps, the largest number for any test matrix, and the relative time

$$\rho \equiv \frac{T_{AS} * \text{average-sweeps (AS)}}{T_{BL} * \text{average-sweeps (BL)}}$$

are shown.

431-10

Table 1. Comparison of Brent-Luk sweeps and ASsweeps

				Averages		Maxima		
p	q	n	trials	AS	BL	AS	BL	ρ
2	2	8	320	3.98	4.33	5	5	1.18
2	4	16	160	5.10	5.38	6	7	1.33
2	8	32	80	6.18	6.29	7	7	1.43
4	2	16	160	4.80	5.40	5	6	1.24
4	4	32	80	5.99	6.31	7	7	1.50
4	8	64	20	7.05	7.55	8	8	1.57
8	2	32	80	5.25	6.28	6	7	1.21
8	4	64	10	6.60	7.60	7	8	1.45
16	2	64	20	6.00	7.30	6	8	1.21

Evidently one ASsweep is more effective in reducing nonorthogonality than one Brent-Luk sweep. This is not surprising, since more orthogonalizations are performed. They are, however, roughly 20-60% less cost-effective.

In order to gauge the reliablility of the statistics generated by this experiment, the standard deviations of the sampled data were measured. In all cases, the standard deviation was less than 0.5. For the samples of size 80 or more, the standard error of the mean is therefore no more than 0.06, so these statistics are quite reliable. For the samples of size 20 and 10, the data are unlikely to be in error by more than 10%

Method B

Method A suffers some loss of speed because, in an ASsweep, some column pairs are generated more than once. By making a small modification to the Brent-Luk array, we can simulate a new supersweep, called an ABSsweep, during which every column pair is generated exactly once.

Figure 3 shows the modified array. The connection from processor 1 to processor p is new. A ring-connected set of processors can easily simulate this structure. The array is still able to perform a Brent-Luk sweep over a set of 2p columns. But it can also perform a second type of sweep, called an ABsweep, that we now describe.

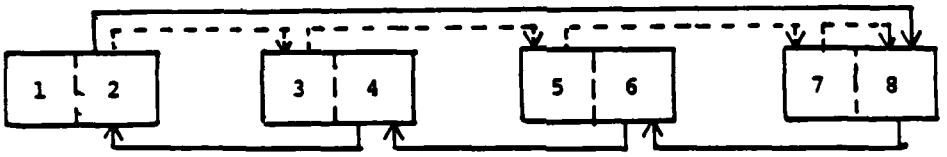


Figure 3. The modified SVD array; n = 8

In an ABsweep, a pair (A,B) of Scolumns (each having p columns) is loaded into the array. During the sweep, all pairs (a,b), a ε A and b ε B are generated exactly once. But no pairs from A×A or B×B are generated.

To implement an ABsweep, place the columns of A in the p left memories and the columns of B in the p right memories of the individual processors. (The set of left (resp. right) processor memories is the Sprocessor's left (resp. right) memory.) Processors do just what they did before: orthogonalize their two columns. Between cycles, A remains stationary, while B rotates one position using the connections shown as solid lines in Figure 3.

An ABSsweep is this. There are 2q Scolumns of p columns each. The initial configuration is as for an ASsweep. During the first Scycle, every Sprocessor performs a Brent-Luk sweep on the 2p columns in its memory. On subsequent Scycles, All Sprocessors perform ABsweeps. Between Scycles, Scolumns move as before.

It is easy to see that in an ABSsweep every column pair is generated exactly once. Thus this scheme implements a true cyclic Hestenes method. The permutation differs, nevertheless from that used by a Brent-Luk array.

Again we have compared the new scheme to the Brent-Luk scheme by an experiment. The experimental set up was the same as for the previous experiment. The results are shown in Table 2. Evidently ABSsweeps are as effective as Brent-Luk sweeps. The standard deviation of the number of ABSsweeps needed was, in all cases, less than 0.5.

Table 2. Comparison of Brent-Luk sweeps and ABSsweeps

				Averages		Maxima	
p	q	n	trials	ABS	BL	ABS	BL
2	2	8	320	4.32	4.33	5	5
2	4	16	160	5.35	5.38	6	7
2	8	32	80	6.36	6.29	7	7
4	2	16	160	5.36	5.40	6	6
4	4	32	80	6.18	6.31	7	7
4	8	64	20	7.50	7.55	8	8
8	2	32	80	6.13	6.28	7	7
8	4	64	10	7.10	7.60	8	8
16	2	64	20	7.00	7.30	7	8

The eigenvalue array

Decomposition of the Brent-Luk eigenvalue array presents the same difficulty, and is amenable to the same solution, as the SVD array. The array of Brent, Luk, and Van Loan for the SVD can also be treated in this way, but there is as yet no experimental evidence to suggest how effective this technique would be.

Acknowledgements

The author wishes to thank Erik Tidén and Björn Lisper for helpful discussions. This research was performed while the author was a guest of the Royal Institute of Technology, Stockholm, Sweden. It was partially supported by the Office of Naval Research under contract N00014-82-K-0703.

References

1. Schreiber, R., "Systolic Arrays: High Performance Parallel Machines for Matrix Computation," Elliptic Problem Solvers, Birkoff, G. and Schoenstadt, A., eds., Academic Press, to appear.

2. Speiser, J.M. and Whitehouse, H.J., "Architectures for Real Time Matrix Operations," Proc. Government Microcircuits Applications Conf., Houston, 1980.

3. Brent, R.P. and Luk, F.T., "A Systolic Architecture for the Singular Value Decomposition," Rept. TR-82-522, Computer Science Dept., Cornell Univ., 1982.

4. Finn, A.M., Luk, F.T., and Pottle, C., "Systolic Array Computation of the Singular Value Decomposition," Proc. SPIE, Vol. 341, Real-Time Signal Processing V, 1982.

5. Heller, D.E., and Ipsen, I.C.F., "Systolic Networks for Orthogonal Decompositions," SIAM J. Scient. and Stat. Comp., Vol. 4, pp. 261-269. 1983.

6. Schreiber, R., "A Systolic Architecture for Singular Value Decomposition," Proc. 1er Colloque International sur Les Méthodes Vectorielles et Parallèles en Calcul Scientifique, EDF Bulletin de la Direction des Etudes et Recherches, Série C, No. 1, pp. 143-148. 1983.

7. Brent, R.P. and Luk, F.T., "A Systolic Architecture for Almost Linear-Time Solution of the Symmetric Eigenvalue Problem," Rept. TR-82-525, Computer Science Department, Cornell Univ., 1982.

8. Schreiber, R., "Systolic Arrays for Eigenvalue Computation," Proc. SPIE, Vol. 341, Real-Time Signal Processing V, 1982.

9. Bienvenue, G. and Mermoz, H.F., "New Principle of Array Processing in Underwater Passive Listening," VLSI and Modern Signal Processing, Kung, S.Y., Whitehouse, H.J., and Kailath, T., eds., Prentice-Hall 1984, to appear.

10. Owsley, N.L., "High Resolution Spectral Analysis by Dominant Mode Enhancement," VLSI and Moder Signal Processing, Kung, S.Y., Whitehouse, H.J., and Kailath, T., eds., Prentice-Hall 1984, to appear.

11. Brent, R.P., Luk, F.T., and Van Loan, C., "Computation of the Singular Value Decomposition Using Mesh-Connected Processors," Rept. TR-82-528, Computer Science Department, Cornell Univ., 1983.

12. Kung, H.T. and Leiserson, C.E., "Systolic Arrays for (VLSI)," Section 8.3 of Mead, C. and Conway, L., Introduction to VLSI Systems, Addison-Wesley 1980.

13. Gentleman, W.M. and Kung, H.T., "Matrix Triangularization by Systolic Array," Proc. SPIE, Vol. 298, Real-Time Signal Processing IV, 1981.

14. Schreiber, R. and Kuekes, P.T., "Systolic Linear Algebra Machines in Digital Signal Processing," VLSI and Modern Signal Processing, Kung, S.Y., Whitehouse, H.J., and Kailath, T., eds., Prentice-Hall 1984, to appear.

END

FILMED

11-85

DTIC